INTEGRATING A PREPROCESSOR BASED ON UNIFICATION IN THE LABORATORY FOR RAPID REWRITING

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Wei Guo

May 2013

INTEGRATING A PREPROCESSOR BASED ON UNIFICATION IN THE LABORATORY FOR RAPID REWRITING

Wei Guo

APPROVED:

Dr. Rakesh Verma, Chairman
Dept. of Computer Science

Dr. Barbara Chapman
Dept. of Computer Science

Robert S. Cartwright
Dept. of Computer Science, Rice University

Dean, College of Natural Sciences and Mathematics

My sincere gratitude goes to Dr. Rakesh M. Verma, for his guidance and support. And my special appreciation goes to my family Huibing Guo, Jinning Chen, and Qiwei Wang for their constant encouragement and support.

INTEGRATING A PREPROCESSOR BASED ON UNIFICATION IN THE LABORATORY FOR RAPID REWRITING

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Wei Guo

May 2013

Abstract

In normalization, traversing the term to find a match with a rule costs most of the time. This thesis presents the design, implementation, and integration of a unification-based preprocessor into the Laboratory for Rapid Rewriting, LRR, and the current status of LRR. LRR consists of two interpreters: Smaran, which stores the history of all rule applications, and TGR, which stands for Term Graph Rewriter. We have improved the preprocessor and the DS-list significantly and efficiently integrated their latest versions into both components of LRR. The performance of the latest version of LRR on some benchmarks – both favorable and unfavorable – is presented and compared with other interpreters including Maude and Rascal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Efficient Rewriting

Term rewriting is one of the main paradigms of computing with equations. This paradigm is the basis for, on one hand, decision procedures based on canonical forms and, on the other hand, the construction of abstract interpreters for directed equations considered as a programming language [16]. The main implementation goal regarding rewriting itself is to compute a normal form of a term whenever it exists as fast as possible. Three methods exist to implement efficient rewriting according to [15]:

1. Compilation of rewrite rules. The idea is to link an executable function which describes the operation to be performed on the reduced item [15]. It is used, for example, in ASSPEGIQUE [4], RAP [10, 11], and LOG [18].

2. Built-in features. Integrating built-in operations and sorts results in efficiency improvement. However, the disadvantage is that it brings more complex operational semantics. It is used in rewriting system such as OBJ [9], and AXIS [20].

3. Optimal rewriting. The purpose is to find the most efficient reduction strategy in order to obtain the normal form of a term if it exists. A reduction strategy decides which redex, or a group of redexes, of a term has to be reduced. It is implemented in, for example, OBJ2, and OBJ3 [12].

The main goal of this thesis is to improve the efficiency of rewriting by integrating a unification-based preprocessor for rules. We can preprocess the rewrite rules to extract information for speeding up the normalization process. The unification results obtained from the rules by the preprocessor do help in locating the next match faster, more accurately and more productively. We also added a number of optimizations to enhance the improvement. Although we use LRR as the platform to demonstrate our idea, this idea is applicable to any rewriting interpreter. We will describe the design, implementation, optimizations, and performance of the preprocessor in this thesis.

1.2 Preliminaries

This section defines a number of fundamental concepts with regard to term rewriting system (TRS).

Signature: A signature is defined as a set $\mathcal{F}$ along with a function *arity*: $\mathcal{F} \to \mathbb{N}$ where $\mathcal{F}$ is a set of *function symbols*, and $arity(f)$ is called the arity of the function symbol f. Function symbols of arity zero are called *constants*. We use $\mathcal{F}_n$ to denote a set of function symbols with arity n; particularly, we use $\mathcal{F}_0$ to denote a set of constants.

Term: Let V be a countable set of *variables* where $V \cap \mathcal{F} = \emptyset$. We define terms, a set $\mathcal{T}(\mathcal{F}, V)$ of $\mathcal{F}$*-terms over* V to be the smallest set that contains V and has the property that $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, V)$ whenever $f \in \mathcal{F}$, $n = arity(f)$, and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, V)$, where $n \in \mathbb{N}$. The outermost function symbol of t is denoted by $root(t)$. The set of variables appearing as subterms of a term t are denoted by $Var(t)$.

To clarify, in this paper, we let C be $\mathcal{F}_0$ and F be $\mathcal{F}_n$, where $n \geq 1$, $n \in \mathbb{N}$. We usually use s, t for general terms, x, y, z for variables, a, b, c for constants, and f, g, h for functions.

The *size*, $|t|$, of a term t is the number of occurrences of variables and function symbols in t. So, $|t| = 1$ if t is a variable, and $|t| = 1 + \Sigma_{i=1}^{n} |t_i|$ if $t = f(t_1, \ldots, t_n)$, $n \in \mathbb{N}$. The *height* of a term t is 0 if t is a constant or a variable, and $1 + max\{height(t_1), \ldots, height(t_n)\}$ if $t = f(t_1, \ldots, t_n)$, where $n \in \mathbb{N}$.

Position: A position of a term t is a sequence of natural numbers to identify the location of a subterm of t. The subterm of $t = f(t_1, \ldots, t_n)$ at position p, denoted $t|_p$, is defined recursively: $t|_\lambda = t$, where λ is the empty sequence, $t|_k = t_k$, and $t|_{k.l} = (t|_k)|_l$ for $1 \leq k \leq n (n \in \mathbb{N}, l \in \mathbb{N})$ and undefined otherwise [19].

Substitution: A substitution is a mapping $\sigma : V \to \mathcal{T}(\mathcal{F}, V)$ that is the identity on all but finitely many elements of V. Substitutions are generally extended to a homomorphism on $\mathcal{T}(\mathcal{F}, V)$ in the following way: if $t = f(t_1, \ldots, t_k)$, then $\sigma(t) = f(\sigma(t_1), \ldots, \sigma(t_k))$ where $1 \leq k \leq n$ and $k \in \mathbb{N}$.

Term matching: Given two terms s and t, term matching determines if a substitution σ exists such that $\sigma(s) = t$, and computes σ if it exists.

Unification: Two terms s and t unify if a substitution σ exists such that $\sigma(s) = \sigma(t)$.

Rewrite rule: A rewrite rule is a pair of terms, (l, r), usually denoted $l \Rightarrow r$, where r does not contain any variables which do not occur in l. Particularly, if $l \in C$, then r cannot contain any variables. In a rule $l \Rightarrow r$, l is called the *left hand side* (LHS), and r is called the *right hand side* (RHS). Notice that if $l \Rightarrow r$ and $l' \Rightarrow r'$ are rules in some rule set, then we will assume (without loss of generality) that $(Var(l) \cup Var(r)) \cap (Var(l') \cup Var(r')) = \emptyset$. We can apply a rule, $l \Rightarrow r$, to a term, t, if there is a substitution, σ and a subterm t' of t, such that $\sigma(l) = t'$; in this case, t is rewritten to s by replacing the subterm $t' = \sigma(l)$ with $\sigma(r)$. t' is defined as a reducible expression (or *redex*). The process of replacing the subterm $\sigma(l)$ with $\sigma(r)$ is called a *rewrite*, denoted $t \to s$.

The fundamental difference between equations and term rewriting rules is that equations denote equality (which is symmetric) whereas term rewriting systems treat equations directionally, as one-way replacements. Further, the only substitutions required for term rewriting rules are the ones found by pattern matching [16].

Term rewrite system: A term rewrite system is a pair, $(\mathcal{T}, R)$, where R is a finite set of rules and $\mathcal{T}$ is the set of terms over some signature, Σ. When the set of terms is clear from the context, we usually omit it and just refer to R itself as a term rewrite system. A *derivation* is a sequence of terms, $t_1, \ldots, t_n$, such that $t_i \to t_{i+1}$ for $i = 1, \ldots, n-1$, where $n \in \mathbb{N}$; this sequence is often denoted by $t_1 \to t_2 \to \ldots \to t_n$.

We denote the i^{th} rule as $rule_i : lhs_i \Rightarrow rhs_i$;. We define that *the i^{th} step of normalization* is a process that builds a new term t_i by applying $rule_j$ to a subterm of term t_{i-1}, in which $0 < i \leq n, i \in \mathbb{N}$. We use $t_{i-1} \to_{(i,j)} t_i$ to denote the i^{th} step of normalization. Thus, the whole process of normalization can be denoted as a sequence, $t_0 \to_{(1,j)} t_1, \ldots, t_i \to_{(i+1,j')} t_{i+1}, \ldots, t_{n-1} \to_{(n,j'')} t_n$. Terms $t_1, \ldots, t_i, \ldots, t_{n-1}$ are called *intermediate results* [27].

A rule $l \Rightarrow r$ is *left-linear* if every variable appears no more than once in l and a rewrite system is left-linear if all rules are left-linear. Two rules $l \Rightarrow r$ and $l' \Rightarrow r'$ are *overlapping* if a non-variable subterm of l unifies with l'. A rewrite system is *orthogonal* if it is left-linear and non-overlapping.

Termination: Given a TRS R, every term t, after finite rule applications, can always be reduced to a term s to which no more rule can be applied. s is called *normal form*. The TRS is called *terminating*. Termination implies that a normal form exists.

Confluence: Whenever a term t can be reduced to two different terms t' and t'' by following different ways of rule application, there is a common term s to which both terms t' and t'' can be reduced. The TRS is called *confluent*. Confluence implies

a unique normal form whenever it exists. A terminating and confluent TRS is called *convergent*.

Defined symbol: A defined symbol (DS) is a symbol that occurs as the root of some LHS in R. All other symbols are called *constructors*. Note that predefined symbols such as mathematical, relational and set operators, etc., are neither defined symbols nor constructors since they will be evaluated eventually. A *defined subterm* is a subterm that has a DS as the top. An *undefined symbol* is the symbol that is not a DS and an *undefined subterm* is a subterm that has an undefined symbol as the top.

1.3 LRR

At the University of Houston (UH), we have been improving LRR for fast rewriting since 1999 when it was introduced first in [23]. The motivations of LRR are below:

1. Theorem proving and formal verification. We used LRR in a Knuth-Bendix completion procedure [23]. Also, we fulfilled CTL model checking using LRR.

2. A testbed for creative rewriting techniques that are practically fast and efficient. We have implemented different reduction strategies as described in later chapters.

3. A tool for both undergraduate and graduate education. At UH, students use it to learn equational programming in the declarative programming course and to illustrate tree automata in the automata theory course. We added RuleMaker

[29], a graphic interface to LRR to enable students to draw tree automata and finite state string and run them. RuleMaker also enables students to view the results of every step. The benefit of LRR is that students can easily program all kinds of automata while avoiding developing packages for specific automata from scratch.

The previous LRR consists of three reduction methods with varing amount of sharing, Tree, TGR, and Smaran. The DS-list reduction strategy was introduced into LRR 2.0 in 2004 [24]. Details will be discussed in Chapter 2.

The latest LRR 3.0 contains a unification-based preprocessor for rules (**UP**) which collects the unification information between each subterm of each RHS with every LHS before normalization. LRR extracts such information to find matches in normalization. Details will be discussed in Chapter 3 and Chapter 5.

LRR also includes a variant detector that can determine if a new term is an alphabetic variant of an existing term, which is usable with the history option. If so, the appropriate variant of the result computed for the existing term is used for further rewriting instead of starting from scratch [23]. LRR provides a set of commands so that it can be called by other systems for symbolic computation. This currently requires the UNIX message passing mechanism. This feature of LRR was used to develop a system called KBS, which integrated an "off-the-shelf" Knuth-Bendix procedure with an earlier version of Smaran.

LRR contains basic datatypes and predefined identifiers. Supported datatypes are integer, float, Boolean, char, set, and untyped. The only two predefined constants

are $true$ and $false$. Predefined functions are following:

- Set operations including union, intersection, insertion, deletion, membership, getting the n^{th} element, and counting the number of elements.
- Regular arithmetic operators including addition, subtraction, multiplication, division, remainder, increment by one, and decrement by one.
- Comparison operators including greater than, less than, greater or equal, less or equal, equal, not equal and logical operators including and, or, xor, and not.

The efficient integration of these built-in datatypes and functions helps in fast normalization.

The input of LRR is a module file with .m extension representing the rules R and a term file with .t extension representing the given term t_0. Similar to algebraic specification languages like ASF+SDF [22], ELAN [3] and Maude [6] an LRR program is composed from modules. Each module defines its own signature and rewriting rules. A module can import other modules. Terms in LRR are written in prefix form. Operators can also be declared AC in LRR, which supports AC matching of left-linear rules. For example, to calculate Fibonacci numbers, we use following rules:

$$fib(x) \Rightarrow f(>(x,1),x) \tag{1.1}$$

$$f(true,x) \Rightarrow +(fib(-(x,1)),fib(-(x,2))) \tag{1.2}$$

$$f(false,x) \Rightarrow 1; \tag{1.3}$$

A linux version of LRR and some examples can be downloaded from Dr. Verma's website `http://www.cs.uh.edu/~rmverma`. Use following command to run LRR.

./lrr [OPTIONS] ... MODULEFILE TERMFILE

And ./lrr - -help for help.

1.4 Problem Definition

This thesis presents a method to make normalization faster by decreasing the time spent on looking for a match between a redex by traversing the term, and a rule by browsing R. We find that many matches found in normalization happen between an instance of a subterm of a RHS and an LHS. This implies that the LHS unifies with this subterm of the RHS, since their variables are "effectively" disjointed. And if a subterm from a RHS can unify with an LHS, there is a great chance to find a match between the instance of the subterm and the LHS when the instance is built by the RHS [27]. This thesis proves that unification does aid in normalization.

The method contains three parts.

1. To collect unification results, we add UP before normalization starts. UP is based on an almost linear unification algorithm (ALU). The primary challenge in this part is to have efficient data structures. The ALU algorithm has strict requirements on data structures to fulfill its almost linear algorithm. Data structures in LRR are fixed and very complex. We try to limit the introduction of too many new data structures. Most data structures are extensions and

modifications of existing data structures.

2. To help normalization, we integrate the unification results from UP into normalization. To find a match, LRR first tries the information extracted from unification results. Most of the time, unification results lead to successful matches with great accuracy. Extraction and storage of unification results are the main problems of this procedure. LRR extracts information when building the instance of a RHS and stores it in a linked list called the ALU-list. Efficient implementation is crucial. We try to minimize any overhead caused by integration.

3. In an effort to improve the method, we add optimizations including fast prediction in section 5.1, Mutually Exclusive Detection (MED) in section 5.2, Descendant Elimination (DE) in section 5.3, Same Point Elimination (SPE) in section 5.4, Changed Signature Detection (CSD) in section 5.5, the V-list in section 5.6, the recyclable ALU-list in section 5.7, and memory management in section 5.8 to help unification results provide more accurate information for matches.

1.5 About the Thesis

Following the introduction of preliminaries and LRR, in Chapter 2 we describe some background information of LRR as a basis for the rest of the thesis and for future reference as well. In Chapter 3 and Chapter 4, we describe UP and its integration to normalization including the data structures and primary algorithms. We describe all

system optimizations in Chapter 5. In Chapter 6, the results and the performance are discussed. Finally we summarize the thesis with some promising directions for future research.

Chapter 2

Background

In this chapter, we will discuss the background knowledge pertaining to the remainder of the thesis including the LRR system, reduction methods and strategies in LRR, as well as data structures and algorithms used in LRR. The latest LRR 3.0 is implemented in C on the Linux platform. It consists of 18 source files and 15 header files. There are three main reduction methods and six reduction strategies among which are the efficient DS-list and ALU-list.

2.1 The LRR System

LRR provides many options. The main options are:

- **-h** lists all options.
- **-ms** uses Smaran reduction method [DEFAULT].

- **-mt** uses TGR reduction method.
- **-mp** uses Tree reduction method.
- **-so** uses original reduction strategy [DEFAULT].
- **-st** uses innermost DS-list reduction strategy.
- **-ss** uses outermost DS-list reduction strategy.
- **-uo** uses the ALU-list reduction strategy.
- **-uv** uses the ALU-list reduction strategy with the V-list.

Different combinations of these options enable different reduction methods and strategies. For example, to normalize using Smaran with the DS-list and the ALU-list without the V-list, we use -ms -st -uo as options. To normalize using TGR with the ALU-list and the V-list without the DS-list, we use -mt -uv.

The source files and header files related to the thesis and their main roles are listed below.

- **alu.h** defines all function prototypes used in alu.c and several data structures used in alu.c.
- **defined_symbols.h** defines all function prototypes used in defined_symbols.c.
- **error.h** defines all error messages.
- **dslist.h** defines all function prototypes used in dslist.c and several data structures used in dslist.c.

- **extern.h** defines all external references of global variables in global.h.
- **global.h** defines most global variables.
- **prep.h** defines data structures used in prep.c.
- **stats.h** defines all function prototypes used in stats.c.
- **tree.h** defines most global constants and variables.
- **alu.c** contains most functions related to UP and the ALU-list. The functions that implement unification and normalization will be further discussed in Chapter 3 and 4. The functions that implement optimizations will be further discussed in Chapter 5. Besides, there are some debugging functions.
- **defined_symbols.c** contains functions that determine whether a subterm is a defined subterm.
- **dslist.c** contains functions that implement the DS-list including initialization, insertion, and deletion.
- **init.c** contains functions that initialize the main heap, the free lists, and hashing functions based on the amount of available memory.
- **main.c** primarily contains the main function, the central loop of LRR, the primary display functions, and the parsing functions for command line arguments.
- **prep.c** contains parser functions for the module file.
- **rec_trs2.c** primarily contains functions needed for normalization in all reduction methods and strategies except for the ALU-list, such as functions that

determine whether two subterms match, functions that apply a rule to a subterm, functions that evaluate predefined operations, and functions that implement various reduction methods and strategies.

- **set_def.c** contains three functions: a) the function inserting a signature into a class, b) the function unioning two classes, c) the function displaying classes for debugging.
- **stats.c** contains functions for advanced statistics.
- **tree.c** contains functions for the 2-3 tree used to store and look up signatures.
- **util2.c** contains various utility functions including functions that parse the rules and display functions.

2.2 Reduction Methods and Strategies

Three reduction methods are listed below.

- Tree is a pure-tree interpreter sharing no common subterm. The given term, intermediate results and the normal form are stored in expression trees. Tree is the slowest algorithm in LRR. It is used as a reference point and for the applications in which the semantics of the rules would be affected by sharing.
- TGR is a term graph interpreter sharing subterms that match different occurrences of the same variable. DAGs are used to represent $t_0, t_1, ..., t_n$ to enable sharing.

- Smaran is a term graph rewriter sharing all common subterms. It stores or tables the history of its reductions, based on the congruence closure normalization algorithm (CCNA). Smaran treats each rule as an equation. Thus, it keeps equivalence classes of terms. Terms are represented implicitly via signatures and each class requires, at most, one special signature which is called the *unreduced signature* of the class. For more details on CCNA, including theoretical justification, please consult [28, 25, 2]. The tabling component of LRR is useful in applications involving certain kinds of nonterminating systems including fixed-point computations, dynamic programming, and retracing/debugging. To illustrate the algorithm, we use the Fibonacci calculator at the end of section 1.3 as an example.

Smaran starts by constructing the signature s of t_0. Then s is inserted into a class and marked the unreduced signature of the class. Smaran tracks the number of this class throughout the process of reduction. Signatures of terms are constructed from the bottom up. We number the rules in a top-down order for convenience and use the symbol '*' to indicate unreduced signature. $t_0 = fib(2)$.

The initial set of classes is:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle^*\}$$

In the 1^{st} step of normalization, LRR calls matching function to find a match between the unreduced signature of any class and the LHS of any rule and a match between class 1 and lhs_1 occurs. While building the instance of the RHS rhs_1, a signature representing the instance $\sigma(rhs_1)$ is created and inserted into class 1

as its unreduced signature. Here we do not show signatures related to the built-in datatypes that can be evaluated directly, and LRR also does not store them in equivalence classes. At the end of 1^{st} step, the classes are below:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle^*\} \quad 2 : \{1^*\} \quad 3 : \{true^*\}$$

In the 2^{nd} step, class 1 matches lhs_2. The instance of rhs_2 is $fib(1) + fib(0)$ which cannot be evaluated. The signature representing this is constructed, inserted into class 1, and marked as its unreduced signature. At the end of 2^{nd} step classes are below:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 4,\ 6\rangle^*\} \quad 2 : \{1^*\} \quad 3 : \{true^*\}$$

$$4 : \{\langle fib\ 2\rangle^*\} \quad 5 : \{0^*\} \quad 6 : \{\langle fib\ 5\rangle^*\}$$

In the 3^{rd} step, class 4 matches lhs_1, and in the 4^{th} step, class 4 matches lhs_3. The term $fib(1)$ which is represented by class 4 reduces to 1 represented by class 2. Thus, class 4 and class 2 are merged into, say 2. The classes at the end of the 4^{th} step are below:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 2,\ 6\rangle^*\} \quad 2 : \{\langle fib\ 2\rangle,\ \langle f\ 7\ 2\rangle,\ 1^*\}$$

$$3 : \{true^*\} \quad 5 : \{0^*\} \quad 6 : \{\langle fib\ 5\rangle^*\} \quad 7 : \{false^*\}$$

Note LRR has updated the signatures which contains class 4 to contain class 2. After the 5^{th} and the 6^{th} steps, the term $fib(0)$ which is represented by class 6 reduces

to 1 represented by class 2. Thus, class 6 and class 2 are merged, say into 2. Now we have:

$$0 : \{2^*\} \quad 1 : \{\langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 2,\ 2\rangle^*\}$$

$$2 : \{\langle fib\ 2\rangle,\ \langle fib\ 5\rangle,\ \langle f\ 7\ 2\rangle,\ \langle f\ 7\ 5\rangle,\ 1^*\} \quad 3 : \{true^*\} \quad 5 : \{0^*\} \quad 7 : \{false^*\}$$

The unreduced signature of class 1 can be evaluated to term 2, which is in class 0. Thus, class 1 and class 0 are merged, say into 0. At the end of the 6^{th} step, we get:

$$0 : \{2^*,\ \langle fib\ 0\rangle,\ \langle f\ 3\ 0\rangle,\ \langle +\ 2,\ 2\rangle\} \quad 2 : \{\langle fib\ 2\rangle,\ \langle fib\ 5\rangle,\ \langle f\ 7\ 2\rangle,\ \langle f\ 7\ 5\rangle,\ 1^*\}$$

$$3 : \{true^*\} \quad 5 : \{0^*\} \quad 7 : \{false^*\}$$

No more matches are found in LRR. Hence, Smaran checks for the existence of a normal form of t_0. The unreduced signature of class 1 is 2 which is irreducible. Therefore, the normal form of $fib(2)$ is 2. Note that LRR needs no more computation to reduce $fib(fib(2))$ because it is represented by the signature$\langle fib\ 0\rangle$ in class 0. Its normal form is also 2. On the other hand, an interpreter that does not store history would calculate $fib(2)$ twice to get the normal form. The compact data structure helps exploit the advantages of storing history and can also speed up normalization [26].

Six reduction strategies in version 3.0 are below.

- The original reduction strategy for Smaran and TGR on a successful match immediately attempts to reduce the instance of the RHS and its descendants.

If LRR finds no match, it backtracks all the way to the root of the intermediate result.

- The leftmost-outermost reduction strategy for Tree is a pure strategy which upon a successful match does not attempt to reduce the instance of the RHS and immediately backtracks to the node m level up, where $m = min(max\{height(lhs)|lhs \rightarrow rhs \in R\}, level\ of\ rhs\ instance)$. It correctly implements leftmost-outermost in pure-tree but not necessarily outermost in TGR or Smaran due to sharing.

- The hybrid version of original Smaran and leftmost-outer reduction strategy for Smaran attempts to reduce the instance of the RHS but not its descendants prior to backtracking.

- The DS-list reduction strategy is based on a list of DSs, the DS-list. Since every match happens between a redex and an LHS, any subterm s such that $root(s) \notin DS$ never matches. Thus, in normalization, LRR builds the DS-list to help find the match. The DS-list always keeps a pointer to the *current* node which is moving from the current node to the next, trying to match the term represented by the node. When rewriting the term, LRR updates the list by adding pointers to any new resultant subterms with DSs as the roots, and by deleting pointers to any obsolete nodes representing the terms that were erased by normalization.

- The ALU-list reduction strategy controls the reduction based on results from

UP which unifies every subterm in every RHS with every LHS. In the ALU-list, a top pointer is maintained. During normalization, the ALU-list pops the top node, trying to match the term represented by the node with some, not all, LHSs according to the unification results. Upon a successful match, LRR updates the ALU-list by adding pointers to new resultant subterms according to the unifications, and deleting pointers to any stale nodes. Details will be discussed in Chapter 4.

- The combination of the DS-list and the ALU-list reduction strategy gives the ALU-list strategy higher priority than the DS-list strategy. During reduction, a subterm in the instance of a RHS is checked by the ALU-list strategy before the DS-list strategy. Terms deleted from the ALU-list are checked by the DS-list strategy. Only when the ALU-list is empty does the DS-list strategy take over the reduction.

2.3 The Data Structures and Algorithms in LRR

LRR stores the rules into an efficient two dimensional array $rule[i][j]$ for quick rule indexing. The rules are indexed by the LHS of rules. i is the unique number that is assigned to each DS. The row $rule[i]$ stores all LHSs which have the same top DS with the unique number i. The pointer to the j^{th} rule with unique number i is stored in $rule[i][j]$.

Constructor Normal Form (CNF) are defined as follows: i) all constructor constants are CNFs; ii) all variables are CNFs; iii) $f(t_1, \ldots, t_n)$ is a CNF if f is a

constructor and $t_1, \ldots, t_n$ are CNFs. CNF represents a subclass of normal forms. The detection of CNF is easier than the detection of normal forms. We introduce CNF because it cuts down unproductive traversals of intermediate results during normalization. No matching attempt is needed below a CNF. Please find details in [26].

LRR uses the structure "RHS" to implement all RHSs, and all the subterms in $t_0, t_1, ..., t_n$ because $t_1, ..., t_n$ are built from RHSs. Figure 2.1 shows the structure "RHS". Field "Category" tells us whether the subterm is a variable, a constant, or a function. Field "Class" stores the class number of a class used in Smaran. Field "Sig" stores the pointer to the term. Field "parameters" stores the pointers to the parameters of a function.

```
Structure: RHS
  USHORT                    label;
  BOOL                      op_flag;
  USHORT                    Category;
  BOOL                      DontReduce;
  UINT                      Class;
  signature                 *SPtr, *Sig;
  UINT                      arity;
  UINT                      RHash;
  struct RHS                **parameters;
  struct RHS                *next,   *next_ptr;
  Value                     val;
  BOOL                      Special;
  RVAR                      *rhs_var;
  UINT                       no_of_var;
  struct Match_Tree_Node    *Ptr;
  UINT                       intrep;
  BOOL                       defined;
  int                        DST_Index;
```

Figure 2.1: The structure "RHS"

LRR uses the structure "RHS_Level" to allow building an instance of a RHS from

the bottom up. When parsing each RHS, LRR links the pointers to the subterms that are at the same depth (level) in a singly linked list. Each level has its own list. Next, LRR links all the headers of these singly linked lists in a doubly linked list. Figure 2.2 shows the structure "RHS_Level". Field "Ptr" is used to build the singly linked list. Fields "Up" and "Down" are used to build the double-linked list between different levels. Field "next_ptr" is used to build the free list. Figure 2.3 shows the RHS of rule (1.1) in the Fibonacci calculator. We use fine dashed arrows to represent the links from the structure "RHS_Level".

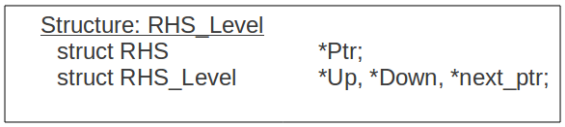

Figure 2.2: The structure "RHS_Level"

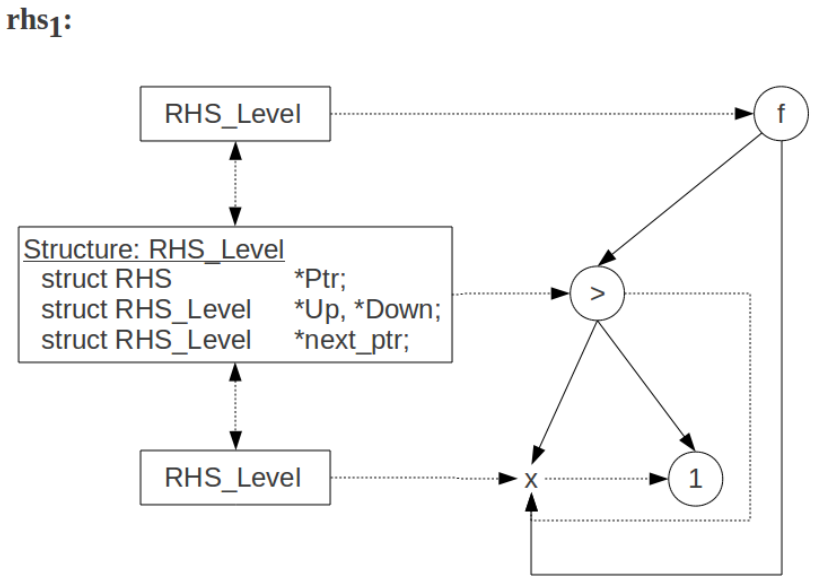

Figure 2.3: An example of the structure "RHS_Level"

LRR uses four families of routines to implement normalization. Each family has variants in Smaran, TGR, and Tree. Since Smaran is the default reduction method with high efficiency, we use regular name for functions related to Smaran. We add suffix "G" for functions related to TGR and Tree. Four families are listed below.

- The normalise family of functions including "normalise32" and "normalise32G" are responsible for rewriting the intermediate results, evaluating the predefined operations, and calling the routing family of functions according to the reduction methods and strategies.

- The routing family of functions including "smaran_reducible32", "smaran_reducible32G", "DSL_reducible" and "DSL_reducibleG" route the control of normalization based on the reduction strategies and call reducible family of functions to look for a match. The original reduction strategy calls functions "smaran_reducible32" and "smaran_reducible32G" which backtrack to the root of intermediate results if no match can be found in or below the instance of the RHS. The DS-list reduction strategy calls functions "DSL_reducible" and "DSL_reducibleG" which traverse the DS-list looking for a match.

- The reducible family of functions including "nr_reducible32", "nr_reducible32G" that traverse the input term to look for a successful match and "nr_reducible32_DND", "nr_reducible32G_DND" that match the input term directly without traversing. They call match family of functions to match two terms.

- The match family of functions "nr_match_further32", "nr_match_futher32G" return true if two input terms match. They also update variable instantiations.

Functions in a family share the same algorithm. The main difference is that TGR works on the pointers to the subterms while Smaran works on the signatures, classes and unreduced signatures.

2.4 The Previous DS-list

The DS-list cuts down the unnecessary traversal of the expression graph by keeping track of the defined subterms. As the given term t_0 is parsed, the DS-list is initialized to have pointers to t_0's defined subterms. During normalization, the current pointer moves from the current node to the next, attempting to match the term indicated by the node. Upon a successful match, the list is updated. When building the instance of a RHS, pointers to the new defined subterms in the instance are inserted into the DS-list. Also, pointers to the stale terms that were deleted by normalization are removed. For example, consider that the expression $f(g(a), b)$ reduces in one step to a, where f, g, b are defined symbols and a is a constructor symbol. Pointers to the subterms $f(g(a), b)$, $g(a)$, and b should be removed from the DS-list, since these defined subterms have been erased. After this update procedure is complete, normalization continues traveling around the DS-list, attempting matches. Normalization completes when the DS-list is empty or when a complete traversal around the list ends without any matches. Since the list can grow or shrink when a match is found, the efficient detection of a complete traversal around the list without any

match requires tracking whether any insertions were made into the DS-list or not.

Previous integration of the DS-list into Smaran implements the DS-list on a dedicated list. Each node in the DS-list contains a pointer to a class whose unreduced signature is labeled by a DS in the current term being reduced.

As the given term is parsed, the existing code calls the function to insert the signatures into classes. This initializes the DS-list to contain pointers to classes of defined subterms occurring in the given term.

Since terms are represented by classes, it is possible to simply track the creation, modification, and union of classes to determine the operations to be performed on the DS-list. Monitoring the classes for three simple conditions is the only measure necessary to maintain the list. First, if the unreduced signature of a newly created class is a defined signature, then the class should be inserted into the DS-list. Second, when inserting a new unreduced signature which is a constructor into a pre-existing class and that class number is marked *current* in the DS-list, this class is deleted from the list. Third, when unioning two classes, if the unreduced signature of the resultant class is labeled by a constructor, then the pointer to the class whose unreduced signature was reduced is removed from the DS-list. Deletions during (cascading) unions may occur anywhere within the DS-list and therefore could require a costly linear search through the list. To avoid repeated searching it is best to batch the deletions. Finally, classes containing defined unreduced signatures representing defined subterms that are erased during a reduction could also be deleted from the DS-list, but this can be even more expensive to determine than deletions caused by cascading unions. For this reason, we chose not to implement this last deletion

condition. Since the only time a class is added to the DS-list is when it is created, and since classes are shared, the DS-list naturally contains only one reference to each class and so terms are not repeated.

The idea of integration into TGR is similar to the idea of integration into Smaran. The main difference is that the DS-list is implemented on top of the DAG data structure of the intermediate result itself by updating the links to subterms. However, this integration was not fully completed in previous LRR. We complete the integration of the DS-list into the TGR and add some optimizations which will be discussed in Chapter 5.

2.5 Rascal and Maude

We compare the performance of the latest LRR with two other rewriting systems: Rascal and Maude.

Rascal is the successor of the ASF+SDF Meta-Environment which is an interactive integrated programming environment supporting the development of ASF+SDF specifications. ASF+SDF is a general-purpose, executable, algebraic specification formalism [21]. It's a combination of the Algebraic Specification Formalism (ASF) which defines conditional rewrite rules and the Syntax Definition Formalism (SDF)

which defines the concrete syntax of a language. ASF+SDF allows the syntax defined in the SDF part of a specification to be used in the ASF part, thus supporting the use of user-defined syntax when writing ASF equations. Its main application areas are the definition of the syntax and the static semantics of (programming) languages, program transformations and analysis, and for defining translations between languages [21]. More details regarding ASF+SDF are available at `http://www.meta-environment.org/`. Rascal is intended to provide more features besides all features of ASF+SDF. Rascal is a new language for meta-programming with the intention to be an engineering tool for programmers who need to build meta programs. Please go to `http://www.rascal-mpl.org/` for Rascal system and related information.

Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications [5]. Influenced by OBJ3 [13], Maude contains OBJ3 as a sublanguage and extends its order-sorted equational logic [14] to membership equational logic [17] which includes sorts, subsorts, operator overloading, and partiality definable by memebership and equality conditions. Maude is implemented in C++ containing a rewrite engine which is a highly modular semicompiler. The semicompiler compiles the most time-comsuming run-time tasks into a system of decision diagrams and automata and interprets the system at run time. In general, the applications of Maude exploit the good features of rewriting logic as a semantic framework and as a logical framework [5]. One important application of logical framework is to use Maude to produce

other formal tools. Applications of semantic framework contain formal specification of architectural description languages, object-oriented designs, and distributed middleware. Maude website is at `http://maude.cs.uiuc.edu/`.

We will discuss the new improvement including UP in Chapter 3, integration of UP in Chapter 4 and optimizations in Chapter 5.

Chapter 3

The Preprocessor

This chapter will discuss in detail how unification helps in normalization and how UP is implementated in LRR. A successful unification between a subterm of a RHS and an LHS has a great chance to predict a match in normalization between the instance of the subterm and the LHS. Before normalization, UP determines and stores these unifications which will be extracted and stored in the ALU-list later in normalization. LRR uses the ALU-list to find matches in normalization, which will be discussed in Chapter 4.

3.1 How Does Unification Help in Normalization

We notice the fact that since $s = \sigma(r|_p)$, an instance of a subterm $t = r|_p$ in a RHS r shares the same overall structure as t, if t unifies with an LHS l, s stands a great chance to match l. Thus, normalization should try s and l directly instead

of looking for a match by traversing all subterms of an intermediate term and all the rules, which should make normalization more efficient by cutting off the time to find matches. To illustrate this, consider the unification result and two steps of normalization in Figure 3.1 below. In the i^{th} step of normalization, $t_{i-1} \rightarrow_{(i,j)} t_i$, a match happens between a subterm u of t_{i-1} and lhs_j. Then the subterm u rewrites to $v = \sigma(rhs_j)$, the instance of rhs_j, and we get t_i. The term v shares the same overall structure as rhs_j and a subterm $x = rhs_j|_p$ unifies with lhs_k. Thus, there is a great chance to find a match between the term $w = v|_p$, the instance of term x, and lhs_k in the next step. In the $i+1^{th}$ step, normalization can try w and lhs_k first. If a match is found, w rewrites to $\sigma'(rhs_k)$.

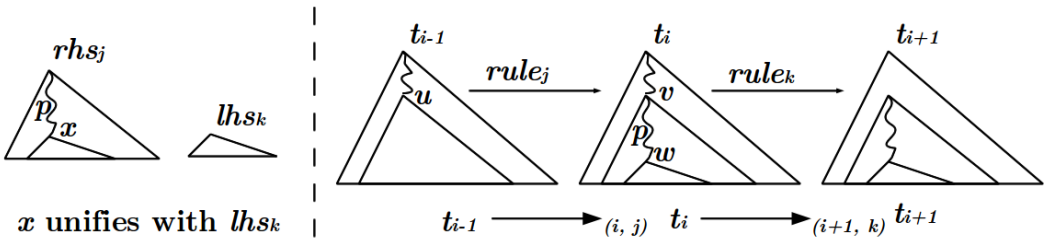

Figure 3.1: Unification results can help in normalization

Actually, significant parts of all the intermediate results, $t_1, \ldots, t_i, \ldots, t_{n-1}$ and the normal form t_n are constructed from RHSs and much of the overall structure of subterms can be safely predicted from the RHSs except for the variable instantiations and unexplored parts of the intermediate terms [27]. Hence, with the unification results, normalization can find matches efficiently. However, not every successful unification result guarantees a successful match. In this case and for normalizing t_0, normalization must revert to ordinary reduction methods and strategies to find matches, such as TGR, Smaran and the DS-list in LRR.

3.2 The Preprocessor

UP attempts to unify every subterm in every RHS with every LHS and stores the successful results.

To store the unification results, we define the *point* P to be the position of the subterm that unifies with at least one LHS, and the *candidate* C to be the rule having the LHS that unifies with a subterm at the point. Thus, a successful unification result can be denoted by a pair (C, P). In the example in Figure 3.1, $rule_k$ unifies with subterm x. $rule_k$ is the candidate, $C = k$ and the position of the subterm x, p is the point, $P = p$. We keep p not the subterm x itself because normalization needs w by following p from v. Just x does not help.

Following p from v to find w is often also time-consuming and storing p can take some space. Thus, instead of storing the real path sequence, we use a flag to mark the subterm that locates at a point in a RHS. Later when normalization rewrites, it traverses the RHS to build the intermediate result. When normalization visits the point in the RHS, it builds the instance of the point in the intermediate result at the same time. In Figure 3.1, when rewriting t_{i-1} to t_i, LRR traverses rhs_j to build v in t_{i-1}. When LRR reaches x in rhs_j, it also builds w which can be used to find next match. Thus, marking every point and keeping its candidate(s) stores successful unification results more efficiently. In this thesis, we still use the conceptual pair (C, P) to refer the successful results. Since one point can unify with more than one LHS, UP uses a singly linked list, the candidate list, to store the pairs for each point.

3.2.1 The ALU Algorithm

UP is using the extended ALU algorithm for unification.

The regular ALU algorithm evolves from unification on term graphs which requires dags implemented in pointer structures to avoid the inefficiency of copying. The overall complexity of the ALU algorithms is almost linear (please see [1] for details). The central idea is to never create new terms but merely to update pointers [1]. The algorithm adds a pointer structure, called instantiation link, to link every variable to its instantiation (if any), which facilitates the substitution to find the instantiation by following the link. The algorithm also requires variables to be shared. UP uses one node for each variable. The algorithm does not require sharing of functions $\mathcal{F}$ (including constants). In dags, we use nodes with circles to represent $\mathcal{F}$, nodes without circles to represent variables V, directed solid edge going from a parent to its child to show parent-child relationship, directed dashed edge going from a variable to its instantiation for instantiation links.

UP needs an acyclic check because not all unifications are *solvable*, or terminated. For example, unifying the term x with term $f(x)$ results in an infinite loop. x is instantiated to $f(x)$, and thus, the subterm x of $f(x)$ can be substituted by $f(x)$. Hence, the x can be substituted by $f(f(x))$. As the substitution continues, x unifies with $f(f(\ldots f(x)\ldots))$. If $x \neq t$ and $x \in Var(t)$, unification of x and t has no solution. In the dag, if $x \neq t$ and $x \in Var(t)$, then there should be one solid path from t to x. If x is instantiated to t, then there is a dashed edge from x to t. Note that there is a circle from t to x including one or more solid edges and one dashed

edge. Thus, a circle indicates that the unification is not solvable. For example, the result of unifying x and $f(x)$ is shown in Figure 3.2 in which there is a cycle from x to $f(x)$. After unification finishes, UP needs to check whether the final directed graph has any circle. If the final directed graph is cyclic, UP does not keep the result.

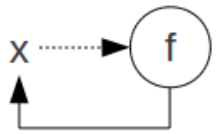

Figure 3.2: One circle found in the unification result

The Extension of the ALU Algorithm allows every term t where $root(t) \in predefined\ functions$ to unify with its possible results. Under strict unification, a function from F (recall $F = \mathcal{F}_n$, where $n \geq 1$) and a constant ($\mathcal{F}_0$) do not unify. It limits the number of successful unifications in rules, which weakens the power of the improvement that unification results bring to normalization. If we allow a function that can be evaluated during normalization and a constant that is one of its possible results to "unify", UP gets more successful unifications and marks more points to facilitate normalization. UP extends the ALU algorithm by allowing comparison operators and logical operators to unify with constants $true$ and $false$, and regular arithmetic operators to unify with integers and floating numbers. To illustrate this, consider the Fibonacci calculator. The RHS of rule (1.1), $f(>(x, 1), x)$, does not unify with any LHS under regular unification. However, it unifies with the LHS of rule (1.2), $f(true, x)$ and the LHS of rule (1.3) , $f(false, x)$ in UP.

This extension brings a problem to normalization by creating unifications that ultimately lead to no match. Evaluation of a predefined function in normalization

returns only one result while UP provides more than one unification pair indicating different possible results. At most one pair leads to a match successfully. Other pairs never succeed in matching. For example, in the Fibonacci calculator, lhs_2 and lhs_3 unify with rhs_1. After being evaluated in normalization, $>(x, 1)$ is either $true$ or $false$. Only one LHS not both may help in finding the next match. It is useless to try the other LHS. We will introduce MED in section 5.2 to fix this problem.

3.2.2 The Data Structures in UP

The two main data structures in UP are the term and the candidate list. To facilitate normalization, UP lets RHSs to carry successful unification results.

3.2.2.1 The Data Structure for terms

We extend the original structure "RHS" to represent terms for unification. LRR uses the structure "LHS" to store LHSs and the structure "RHS" to store RHSs. We chose the structure "RHS" for two primary reasons. Firstly, to help normalization, LRR needs to know which subterm unifies with which LHS, not which LHS unifies with which subterm. So it is more convenient to mark the point and associate a list of candidates to each point when UP browses RHSs. Secondly, when normalization rewrites a term, it needs to traverse the RHS to build its instance. So a RHS can carry unification results to normalization, which saves time and space by not building a separate data structure to carry the results. When parsing an LHS, UP also stores a copy of the LHS into the structure "RHS" so that unification happens between two

terms in the same data structure.

Figure 3.3 shows the latest data structure of the structure "RHS". All fields starting with ALU are created for UP and the ALU-list reduction strategy. Among original fields, field "Category" tells us whether the term is a variable, a constant, or a function. Field "Class" stores the class number for Smaran. Field "Sig" stores the pointer to the term. Field "ALU_para" is a separate list to store parameters of a function, which is dedicated for unification. When LRR parses RHS, it creates two expression trees. One is built on the original field "parameters". The other is built on the field "ALU_para" solely for unification. It is good to have a clean and dedicated data structure for unification in such a complicated program as LRR. The field "ALU_rank" is used to implement union and find operations in the ALU algorithm which will be discussed in section 3.2.3. Fields "ALU_color, ALU_dtime, ALU_ftime are used in depth-first search which is called by the acyclic check. The field "ALU_is" is the instantiation link. P in the pair (C, P) is implemented by the field "ALU_point" indicating whether the location of the term is a point with a default value 0. If the location of a subterm is a point, UP changes the value to 1. The field "ALU_pointcnt" stores how many points are in a RHS. The field "ALU_candidate" points to the candidate list. The fields "ALU_dtime, ALU_ftime" also helps DE in section 5.3 to find the descendants. The fields "ALU_MEid" and "ALU_MEcnt" collect information in UP for MED in section 5.2. The field "ALU_CNF" indicates whether the term is a constructor normal form.

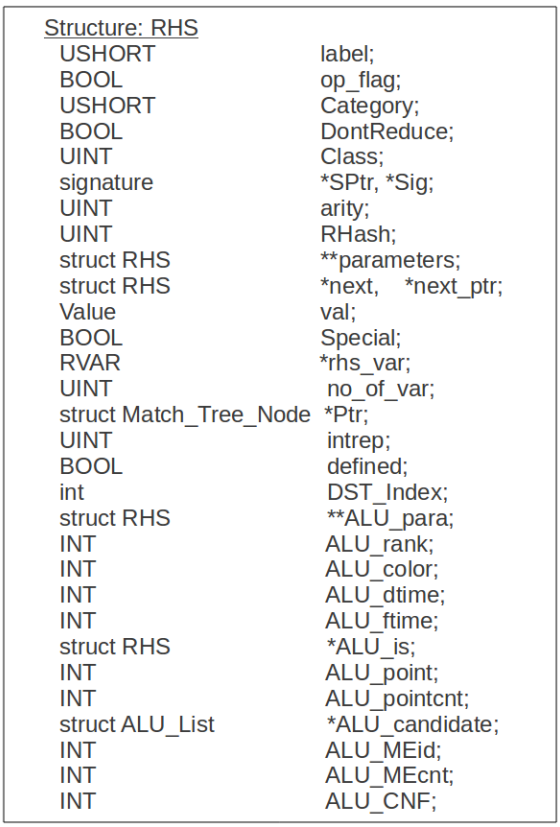

Figure 3.3: The latest structure "RHS"

3.2.2.2 The Data Structures for the Candidate List

The candidate list is a singly linked list to store candidates for one point in a RHS. We use the structure "ALU_List" to store the list. Each node in the list is using the structure "ALU_List_Node". The candidate list is a list of candidates which unify with same P.

The structure "ALU_List" is shown in Figure 3.4. In UP, the field "Head" points to the candidate list. The candidate list contains a "dummy" header which stores no

candidate's information. The storage of the real candidates' information starts from the second node. The field "Size" stores the size of the list except for the header indicating the number of candidates in a list. In Chapter 4, the structure "ALU_List" is also used in normalization to store the ALU-list which needs all the fields.

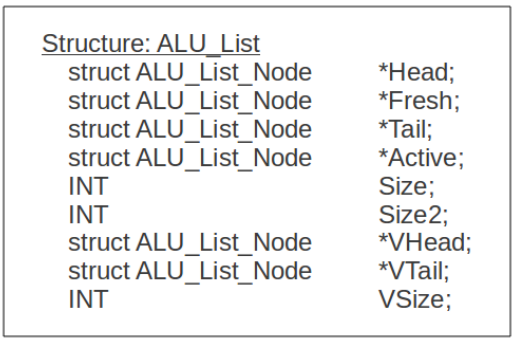

Figure 3.4: The structure "ALU_List"

Figure 3.5 shows the structure "ALU_List_Node" which represents candidates C in UP. Recall in LRR, rules are stored in a two dimensional matrix. Thus, fields "LHSi" and "LHSj" are used to index C. The field "Next" points to the next node in the candidate list. The field "Same" labels each point with a unique number for SPE in section 5.4. Fields "MEVal", "MEValMax" collect the information for MED in section 5.2. We also use the structure to implement the ALU free list in section 5.8. The field "next" points to the next node in the ALU free list. Besides, we use this structure to implement nodes in the ALU-list. Fields "Sig", "Class", "LHSi", "LHSj", "Next", "Loop", "Same", "DFMin", "DFMax" stores necessary information for normalization in Chapter 4. Also, fields "Sig", "Class", "Next" are needed in the V-list in section 5.6.

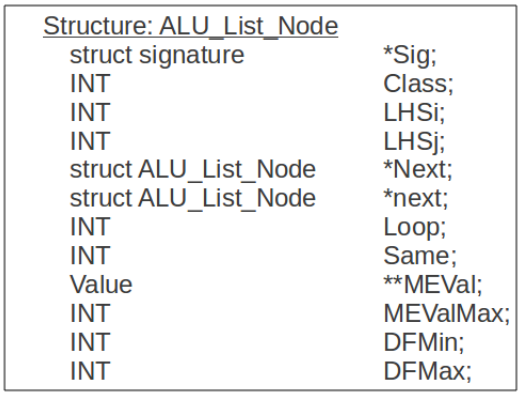

Figure 3.5: The structure "ALU_List_Node"

Figure 3.6 shows a candidate list of a point in a RHS with more than one candidate. Figure 3.7 shows unification results of a RHS. Take the Fibonacci calculator 1.1 as an example. Using the thicker dotted arrow, Figure 3.8 shows the unification results of all rules.

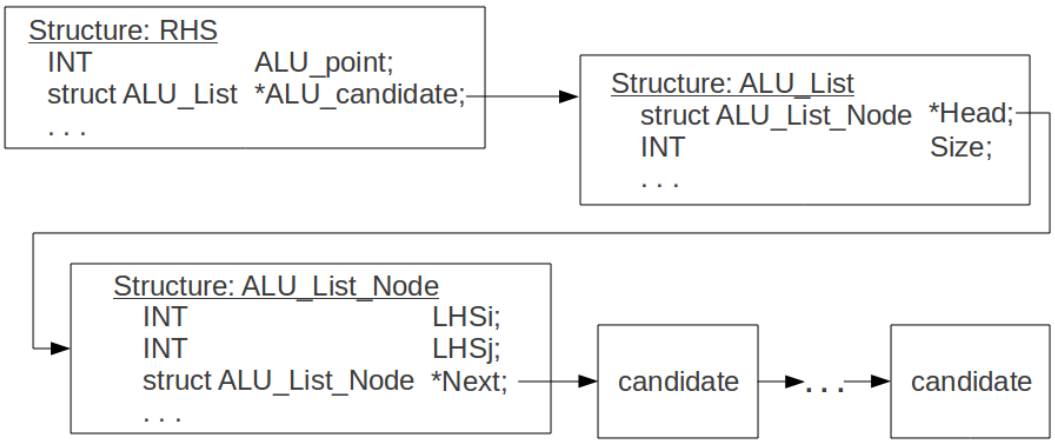

Figure 3.6: A candidate list of a point

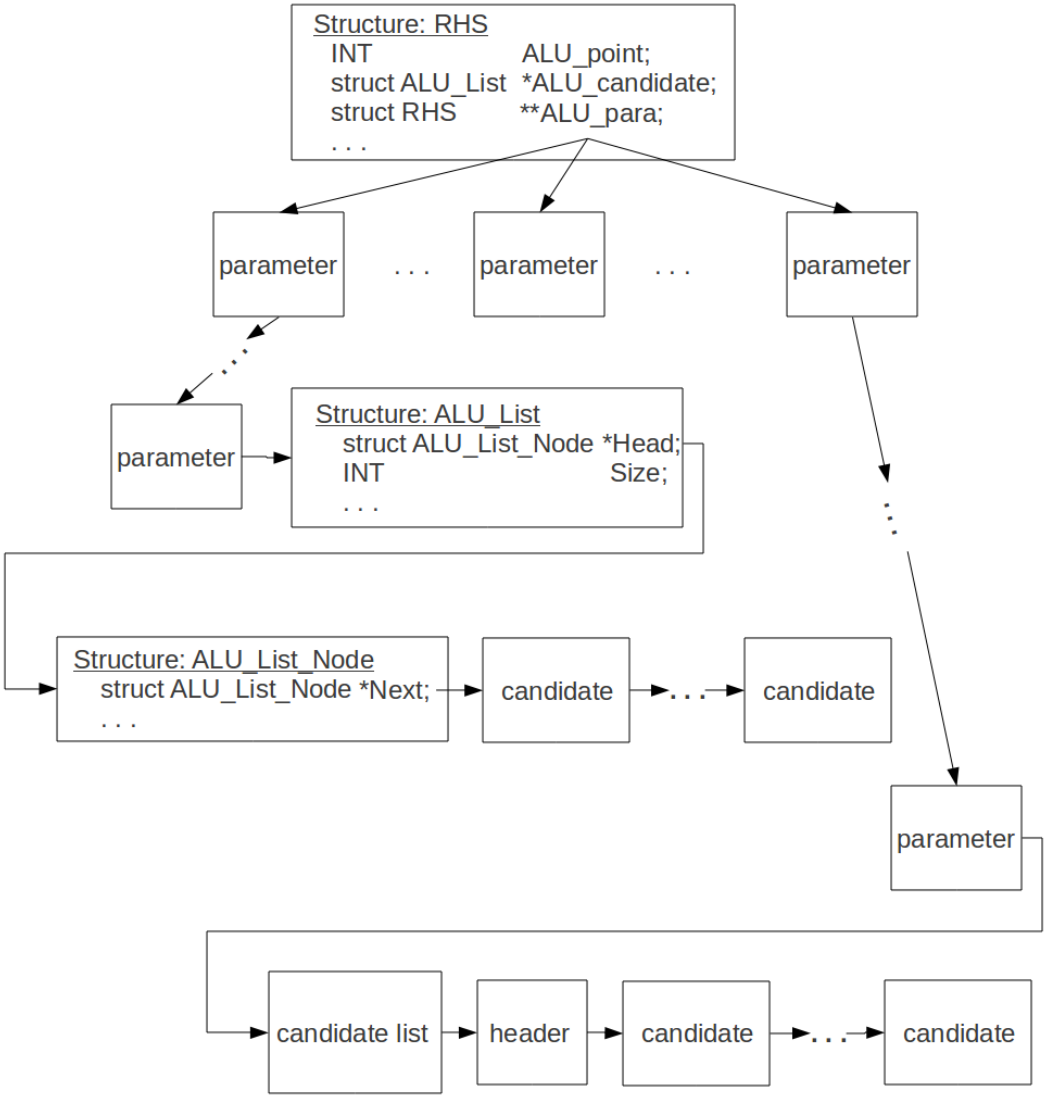

Figure 3.7: Unification results of a RHS

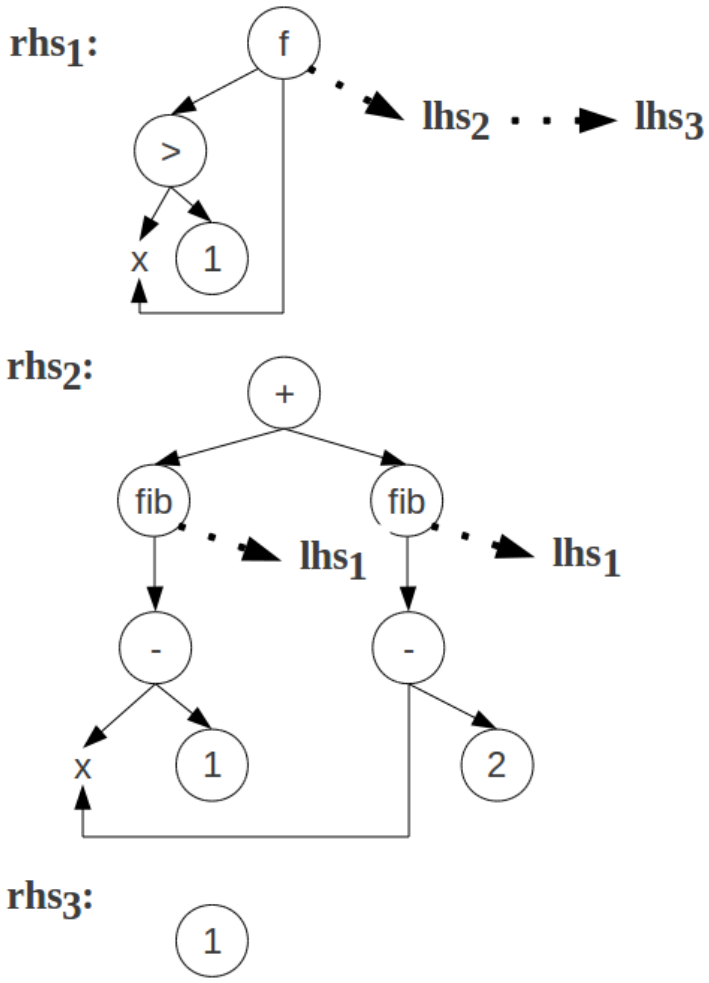

Figure 3.8: Unification results of the Fibonacci calculator

3.2.3 The Algorithms in UP

UP collects the successful unification results to help normalization. The primary algorithm is the extended ALU algorithm which requires subroutines of depth-first search (DFS), union and find operations, and the parenthesis theorem. UP also calls

other functions for the optimizations in Chapter 5. The primary functions and their major purposes are listed below and the call graph of the below functions is shown in Figure 3.9. The functions used for optimizations will be discussed later in Chapter 5. We name functions related to UP and the ALU-list with the names starting with "ALU".

- The non-recursive function "ALUDFLM" tries to unify every subterm in every RHS with every LHS and stores the successful results. It is the main function in UP with no input or output. The function updates global variables representing RHSs by marking every point and associating a candidate list to each point.

- The non-recursive function "ALU" tries to unify two terms and checks if there is any cycle in the unification result. The inputs are two pointers each of which points to a term. The output is true if two terms unify and have no cycle in the resulting dag, and false otherwise.

- The recursive function "ALUunify" attempts to unify two terms without acyclic check. The inputs are two pointers each of which points to a term. The output is true if two terms unify, and false otherwise.

- The recursive function "ALUacyclic" searches a term in a depth-first leftmost order to check if it has any cycle. The input is a pointer to a term. The output is true if there is no cycle, and false otherwise.

- The non-recursive function "ALUno_cycle" checks if a term is in a loop using parenthesis theorem. The input is a pointer to a term. The output is true if the term is not in a loop, and false otherwise.

- The recursive function "ALUDFS" traverses a term in a depth-first leftmost order. The input is a pointer to a term. It has no output. The function updates fields "ALU_color", "ALU_dtime", "ALU_ftime".

- The non-recursive functions "ALUfind_root", "ALUlink", and "ALUunion_root" implement find-union algorithm. The function "ALUfind_root" finds the root of a tree. The function "ALUlink" links two trees and modifies their ranks if needed. The function "ALUunion_root" links the roots of two trees.

- The recursive function "ALUreset" resets the term after unification.

- The functions "ALUME_read", "ALUunify_LHS", and "ALUME_copy" collect information for MED in section 5.2.

- The function "ALUDF_RHS" collects information for DE in section 5.3.

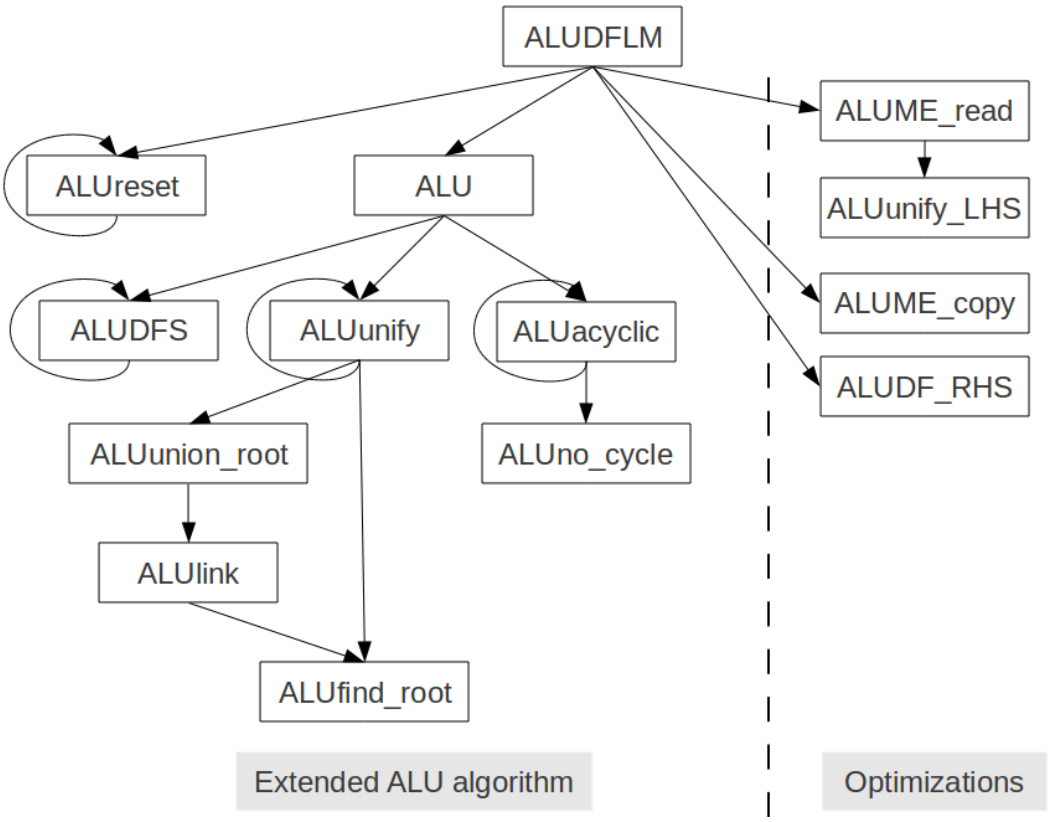

Figure 3.9: The call graph of the primary functions in UP

3.2.3.1 The Function "ALUDFLM"

The function traverses all rules. For each LHS, it traverses every subterm in every RHS. For each RHS, the function uses a local stack to implement non-recursive traversal. Thus, the function unifies every subterm in every RHS with every LHS. If they unify, the function marks the ALU point, updates the candidate list, and resets the subterm and the LHS. Besides, the function updates information for the optimizations in Chapter 5. The pseudocode is below.

```
1: procedure ALUDFLM
2:     ALUME_READ                                                    ▷ for ME
3:     for each rule rule do
4:         lhs = the LHS of rule
5:         rhs = the RHS of rule
6:         ALUDF_RHS(rhs)                                            ▷ for DE
7:         same = 0                                                  ▷ for SPE
8:         r = rhs
9:         while the stack is not empty OR r.Category == function  do
10:            if r.Category == function then
11:                same = same + 1                                   ▷ for SPE
12:                initialize a new candidate list
13:                for each LHS l do
14:                    if ALU(l, r)==true then
15:                        initialize a new candidate candidate
16:                        candidate.LHSi = the first index of l
17:                        candidate.LHSj = the second index of l
18:                        candidate.Same = same                     ▷ for SPE
19:                        candidate.Next = NULL
20:                        collect MED info
21:                        r.ALU_point = 1
22:                        add the new candidate to the candidate list
23:                    end if
```

```
24:                 ALUreset(l)
25:                 ALUreset(r)
26:             end for
27:             for i = 1 → r.arity do
28:                 if r.ALU_para[i] is a function then
29:                     push r.ALU_para[i] into the stack
30:                 end if
31:             end for
32:         end if
33:         if the stack is not empty then
34:             pop top from the stack
35:             r = top
36:         end if
37:     end while
38:     for i = 1 → rhs.arity do
39:         ALUME_copy(rhs.ALU_para[i], rhs.parameters[i])        ▷ for ME
40:     end for
41: end for
42: end procedure
```

The function "ALUDFLM" browses every rule using a for loop in line 3. The while loop starting from line 9 implements the local statck to browse every subterm in one RHS. The fuction calls the function "ALU" to try to unify a term with an LHS in line 14 and calls the function "ALUreset" to reset terms in line 24 and 25.

3.2.3.2 The Function "ALU"

The function implements the entire extended ALU algorithm. It unifies two input terms. Next, it checks if the resultant graph has a cycle. The pseudocode is below.

```
1: procedure ALU(x, y)
2:     if ALUunify(x, y) == true then
3:         ALUDFS(x)
4:         if ALUacyclic(x) == true then
5:             return true
6:         end if
7:     end if
8:     return false
9: end procedure
```

The function "ALU" calls the function "ALUunify" in line 2. If "ALUunify" returns true, the funtion calls "ALUDFS" in line 3 and "ALUacyclic" in line 4 to check if there is any cycle. The function "ALU" returns true if "ALUacyclic" returns true.

In UP, the discovery time and the finishing time are implemented by the fields "ALU_dtime" and "ALU_ftime". The field "ALU_color" in the structure "RHS" has the value of 0 initially, 1 when the vertex is discovered, and 2 when the vertex is finished. The function "ALUDFS" implements the standard recursive DFS updating "ALU_color", "ALU_dtime", and "ALU_ftime" for each vertex.

3.2.3.3 The Function "ALUreset"

The function goes through the RHS recursively in the depth-first order and resets fields that are related to unification. The pseudocode is below.

```
1: procedure ALURESET(x)
2:     x.ALU_rank = 0
3:     x.ALU_color = 0
4:     x.ALU_dtime = 0
5:     x.ALU_ftime = 0
6:     x.ALU_is = NULL
7:     if x is a function then
8:         for i = 1 → x.arity do
9:             ALURESET(x.ALU_para[i])
10:        end for
11:    end if
12: end procedure
```

The function calls itself in line 9 and resets the fields "ALU_rank", "ALU_color", "ALU_dtime", "ALU_ftime" to 0 and "ALU_is" to NULL.

3.2.3.4 The Function "ALUunify"

The Function unifies two input terms t_1 and t_2. It finds the ends of the instantiation chains of the two input terms, say s_1 and s_2. Next it unifies s_1 and s_2 according to the four cases below.

1. If both s_1 and s_2 are variables, the function unions them.

2. If s_1 is a variable and s_2 is not a variable, the function instantiates s_1 to s_2.

3. If s_1 is not a variable and s_2 is a variable, the function instantiates s_2 to s_1.

4. If neither s_1 nor s_2 is a variable, the function checks whether s_1 and s_2 have the same function symbol and their children unify. The function unifies their children recursively. The extension of the ALU algorithm happens when one of s_1 and s_2 is a constant and the other is a built-in function. If the constant is *true* or $false$ and the built-in function returns Boolean value, or the constant is an integer or a float and the built-in function is arithmetic, the function unions them.

Besides, MED collects information, which will be discussed in 5.2. The pseudocode is below.

```
1: procedure ALUUNIFY(t1, t2)
2:     s1 = ALUFIND_ROOT(t1)
3:     s2 = ALUFIND_ROOT(t2)
4:     if s1 == s2 then
5:         return true
6:     end if
7:     switch s1.Category, s2.Category do
8:         case variable, variable
9:             ALUUNION_ROOT(s1, s2)
10:            return true
```

```
11:         case variable, non-variable
12:             s_1.ALU_is = s_2
13:             return true
14:         case non-variable, variable
15:             s_2.ALU_is = s_1
16:             MED collects information
17:             return true
18:         case default
19:             if s_1 and s_2 have same function symbol then
20:                 ALUunion_root(s_1, s_2)
21:                 for i = 1 → s_1.arity do
22:                     if ALUunify(s_1.ALU_para[i], s_2.ALU_para[i])
23:                         == false then
24:                         return false
25:                     end if
26:                 end for
27:                 return true
28:             else
29:                 if s_1 == true or false AND s_2 is a comparison or logical function
    then
30:                     MED collects information
31:                     ALUunion_root(s_1, s_2)
32:                     return true
```

```
33:             else
34:                 if s1 is a number AND s2 is an arithmetic function then
35:                     ALUunion_root(s1, s2)
36:                     return true
37:                 end if
38:                 return false
39:             end if
40:         end if
41: end procedure
```

The function calls the function "ALUfind_root" to find the ends of instantiation chains in line 2 and 3. Four cases are implemented in the switch statement starting from line 7. For case 1, the function calls the function "ALUunion_root" to union them in line 9. For case 2 or 3, the function updates the instantiation links in line 12 and 15. For case 4, the function recursively calls itself in line 22 to unify children. The extension of the ALU algorithm occurs in line 29 and line 34.

3.2.3.5 The Union and Find Operations

The union and find operations are based on the disjoint-set forest with the union-by-rank heuristic [7]. The node in this operation is the node in the dag. The edge is the instantiation link. The instantiation link of the root of the tree points to NULL. Every node maintains its rank and is initialized to a singleton set. The single node contains an initial rank of 0. Initialization is implemented when LHSs and RHSs are parsed. The find operation finds the root of the tree leaving all ranks untouched.

The union operation has two cases: a) If the two roots have unequal ranks, we make the root with lower rank the child of the root with higher rank without changing ranks. b) If two roots have equal rank, we arbitrarily choose one root as the parent and increase its rank by 1. The pseudocode is below.

```
1: procedure ALUUNION_ROOT(x, y)
2:     ALULINK(ALUFIND_ROOT(x), ALUFIND_ROOT(y))
3: end procedure
1: procedure ALULINK(x, y)
2:     if x.ALU_rank > y.ALU_rank then
3:         y.ALU_is = x
4:     else
5:         x.ALU_is = y
6:         if x.ALU_rank == y.ALU_rank then
7:             y.ALU_rank = y.ALU_rank + 1
8:         end if
9:     end if
10: end procedure
1: procedure ALUFIND_ROOT(x)
2:     y = x
3:     while y.ALU_is ≠ NULL do
4:         y = y.ALU_is
5:     end while
6:     return y
```

7: **end procedure**

The instantiation link is implemented by the field "ALU_is" in the structure "RHS" and the rank is implemented by the field "ALU_rank". The function "ALUfind_root" implements the find operation and the functions "ALUunion_root" and "ALUlink" implements the union operation.

3.2.3.6 The Acyclic Check

The acyclic check recursively searches the resulting graph in a depth-first order. On every vertex it checks whether it is in a loop.

For each vertex, the acyclic check detects the cycle using the nesting of descendant's intervals based on the parenthesis theorem. It says that vertex v is a proper descendant of vertex u in the depth-first forest for a directed graph G if and only if $u.d < v.d < v.f < u.f$, where $u.d$ denotes the discovery time of u, and $u.f$ denotes the finishing time of u in the DFS. Please refer to [7] for details. The vertex is in a loop if there is a directed edge from it to its ascendant. Please note that edges consist of instantiation link (dashed) edges and parent-child (solid) edges. In another way, the vertex is not in a loop if it is a variable and is not instantiated to its ascendant, or if it is a function and none of its children are its ascendant. The pseudocode is below.

1: **procedure** ALUACYCLIC(x)
2: $x.ALU\_color = 3$
3: **if** ALUNO_CYCLE(x) == $false$ **then**

```
4:          return false
5:      end if
6:      if x.ALU_is ≠ NULL AND x.ALU_is.ALU_color == 2 then
7:          if ALUacyclic(x.ALU_is) == false then
8:              return false
9:          end if
10:     end if
11:     if x is a function then
12:         for i = 1 → x.arity do
13:             if x.ALU_para[i].ALU_color == 2
14:                 AND ALUacyclic(x.ALU_para[i]) == false then
15:                 return false
16:             end if
17:         end for
18:     end if
19:     return true
20: end procedure
1: procedure ALUno_cycle(x)
2:     if x is a variable AND x.ALU_is ≠ NULL  then
3:         if x.ALU_is.ALU_dtime < x.ALU_dtime
4:             < x.ALU_ftime < x.ALU_is.ALU_ftime then
5:             return false
6:         end if
```

```
7:     else
8:         for i = 1 → x.arity do
9:             if x.ALU_para[i].ALU_dtime < x.ALU_dtime
10:                < x.ALU_ftime < x.ALU_para[i].ALU_ftime then
11:                return false
12:            end if
13:        end for
14:    end if
15:    return true
16: end procedure
```

The function "ALUacyclic" calls the function "ALUno_cycle" on every vertex to check the loop in line 3. It recursively calls itself to search via instantiation link edge in line 7 and parent-child edge in line 14. After the function "ALUDFS", the field "ALU_color" of every vertex has the value 2. Thus, the value of the field "ALU_color" in the function "ALUacyclic" is 2 initially, 3 when the vertex is discovered, and 4 when the vertex is finished. The parenthesis theorem is used in the function "ALUno_cycle" in line 3 and in line 9.

After the function "ALUDFLM", UP has all successful unification information which is carried by RHSs of the rules. The ALU-list reduction strategy in the next chapter will use it efficiently to help normalization.

Chapter 4

Integration into Normalization

In this chapter, we will discuss how LRR integrates the unification results into normalization and how the ALU-list reduction strategy controls the normalization process.

Normalization starts from t_0 and ends in a normal form t_n if it exists. In the i^{th} step, normalization needs to find a match between a subterm s of t_{i-1} and an LHS lhs_j, and then rewrites t_{i-1} to t_i by applying rule $rule_j$ to s. We define the subterm s the *active term* and the instance of rhs_j the *current term*. In another way, the active term in t_{i-1} rewrites to the current term in t_i. $rule_j$ is defined as *the current rule*. The successful unification information collected by UP helps in finding a match by predicting the active term and the current rule for future steps. Recall in Figure 3.1, UP provides that the subterm x in rhs_j and lhs_k unify. Thus, in the $i + 1^{th}$ step, normalization predicts the active term w for the next step, tries to match it with lhs_k and rewrites t_i to t_{i+1} if the match attempt succeeds.

In the i^{th} step where $i > 1$, the integration of the unification results (C, P) into normalization happens when t_{i-1} rewrites to t_i. Recall before normalization, the point P is marked in the RHS and candidates are stored in the candidate list. In the i^{th} step, normalization traverses the RHS to build its instance from the bottom up. It checks every node in the RHS to see if it is a point. If it is, normalization keeps its instance and the candidate(s). We define a *normalization tuple* (i, c, s) in normalization, where i indicates the i^{th} step, c indicates a candidate, in which $c = C$, and s represents the instance of the point P, $\sigma(P)$, which could be the active term. We define that the *normalization tuple* (i, c, s) *matches* if lhs_c matches s. If they match, we can apply $rule_c$ to t_i to get t_{i+1}. In Figure 3.1, when building v, the instance of $rule_j$, normalization traverses rhs_j. When visiting x in the rhs_j, normalization knows it is a point and keeps w, the instance of x, and the candidate $rule_k$ in the tuple (i, k, w). If w matches lhs_k, normalization applies $rule_k$ to w to get t_{i+1}.

Since one RHS can have multiple points and one point can have multiple candidates, there may be multiple tuples in one step of normalization. Normalization uses a singly linked list, the ALU-list, to store the tuples. Also, since not every unification pair (C, P) leads to a tuple (i, c, s) that matches and no unification pair is available for the 1^{st} step, normalization must resort to other reduction methods. We manage TGR, Smaran, the DS-list and the ALU-list to work together efficiently. The ALU-list can neither start nor end normalization. However, it helps normalization in between.

Normalization can be implemented in loops, each of which indicates one step of

normalization that contains two parts: a) finding the active term and the current rule; b) building the instance. The ordinary normalization methods including TGR and Smaran are implemented mainly by a main loop. The 1^{st} step is outside the loop. In the main loop, normalization builds the instance and then looks for the next match by term traversal. If match is found, normalization updates the active term and the current rule for the next step and enters into the next iteration. Normalization using the ALU-list reduction strategy also contains these two parts. However, the differences are: a) normalization finds the active term and the current rule from the ALU-list whenever the ALU-list is not empty; and b) normalization updates the ALU-list when building the instance.

4.1 The ALU-list

The ALU-list is a singly linked list of pointers to normalization tuples. In each step of normalization, upon a successful match, normalization updates the list by inserting pointers to new tuples into the list and deleting stale pointers. Normalization pops the first pointer to get the possible active term and current rule for the next step and tries to match. If the match succeeds, normalization goes to the next step. Otherwise, normalization pops the next pointer. When the ALU-list is empty, normalization resorts to term traversal used in TGR or Smaran or the DS-list to find the next match.

Generally speaking, the ALU-list should predict the next match more precisely than the DS-list or ordinary methods due to the fact that it carries more information.

The comparison is below.

- The ALU-list can provide both the active term and the current rule for the next step.
- The ordinary methods look for the next match by traversing the current term from the top and searching all rules. If no match is found, the ordinary methods traverse the entire intermediate result from the top and search all rules. The ordinary methods cannot predict the next match.
- The DS-list looks for the next match by traveling the DS-list to find the active term t for the next step. It tries to match t, in which $root(t) \in DS$, with a group of rules the LHSs of which have same DS as $root(t)$ as the top symbol. The DS-list provides the active term with a group of rules.

Hence, whenever the ALU-list is not empty, the ALU-list controls the normalization procedure. If the ALU-list is empty and the DS-list is enabled, normalization must go to the DS-list. If the ALU-list is empty and the DS-list is not enabled, normalization must go to the ordinary methods. Also, if the DS-list is enabled and a subterm in an intermediate result is a DS and also an instance of a point, normalization puts it into the ALU-list.

Section 5.7 introduces the recyclable ALU-list which extends the ALU-list to two parts: *the fresh part* and *the recycled part*. All portions of the ALU-list we discussed before refer to the fresh part. The recycled part is used to collect the tuples that fail in matching. When the fresh part is empty, the recycled part becomes the fresh

part. Details will be discussed in section 5.7. We usually refer the ALU-list to the fresh part. We will be specific when we refer the recyclable ALU-list or the recycled part.

Section 5.6 introduces the V-list which is used to store variable instantiations in intermediate results. The V-list is an option when the ALU-list is enabled and the DS-list is not. Normalization goes to the V-list after the ALU-list is empty. Details will be discussed in section 5.6.

Insertions to the ALU-list happen in the following two cases.

- Whenever normalization meets a point when it traverses a RHS to build the instance of the RHS, it creates a tuple for each pair in the candidate list that belongs to the point and inserts the pointer to the tuple into the ALU-list. To save the time inserting and popping the node, normalization holds the first tuple in one step temporarily not in the ALU-list until next step and inserts remaining tuples into the ALU-list. In the next step, normalization tries the first tuple. If match succeeds, normalization continues. Otherwise, normalization pops the top node from the ALU-list. With the help of MED in section 5.2, the first tuple has great accuracy in predicting the next active term and next current rule. Thus, holding the first tuple temporarily in each step cuts the overhead.

- Insertions to the recycled part happen whenever a tuple leads to an unsuccessful match. New pointers to tuples are added directly at the end of the recycled part.

Deletions of the ALU-list happen in the following two cases.

- Normalization pops the top tuple for the next match.
- Optimizations discussed later in Chapter 5 including MED in section 5.2, DE in section 5.3, SPE in section 5.4, and CSD in section 5.5 eliminate tuples from the ALU-list.

4.2 The Data Structures for the ALU-list

The data structures for the ALU-list are similar to the data structures for the candidate list in section 3.2.2.2. The ALU-list is a list of pointers to tuples (i, c, s) while the candidate list is a list of pointers to unification pairs (C, P). The ALU-list reuses the data structures. Each tuple is implemented by the structure "ALU_List_Node" and the list is implemented by the structure "ALU_List".

In Figure 3.4, the field "Head" starts the ALU-list with a "dummy" header. The storage of the real candidates' information starts from the second node. The field "Fresh" indicates the end of the fresh part and the field "Tail" indicates the end of the entire recyclable ALU-list. The field "Active" indicates the insertion position by pointing to the previous node of the insertion position in the fresh part. The field "Size" stores the size of the fresh part excluding the header. The field "Size2" stores the size of the recycled part. The field "VHead" starts the V-list. The field "VTail" indicates the end of the V-list, and the field "VSize" stores the size of the V-list.

The structure "ALU_List_Node" stores one normalization tuple (i, c, s). In Figure

3.5, when Smaran is used, the field "Class" stores the class of s in the tuple and the field "Sig" is a pointer to the term. "Sig" is needed in the CSD in section 5.5. When TGR is used, the field "Sig" is a pointer to the term indicating s. In both Smaran and TGR, fields "LHSi" and "LHSj" are the indexes of c in the matrix of rules R. The field "Next" points to the next tuple. The field "Loop" stores i. The field "Same" is used in SPE in section 5.4. Fields "MEVal", "MEValMax" are used in MED in section 5.2, and fields "DFMin", "DFMax" are used in DE in section 5.3.

Figure 4.1 shows a typical recyclable ALU-list that contains a non-empty fresh part and a non-empty recycled part.

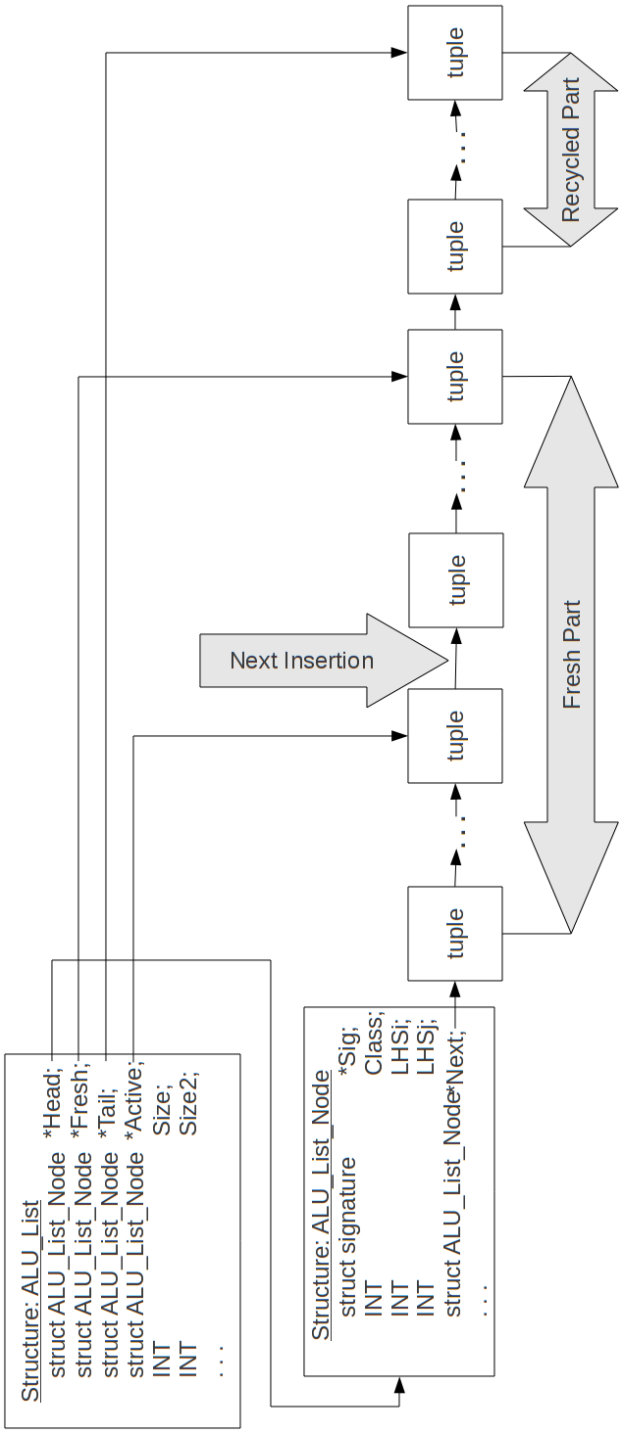

Figure 4.1: The ALU-list

LRR uses the structure "Signature" to represent terms, and the structure "xClass" to represent classes of the intermediate results. "Signature" is used in both Smaran and TGR. "xClass" is used only in Smaran.

The ALU-list, to clarify in which list the term is currently, adds a field "ALU_DAV" in "Signature" and in "xClass". A term can only be in one of the three lists: the DS-list, the ALU-list, and the V-list. The value is 0 if the term is not in any list, 1 if the term is in the DS-list, 2 if in the ALU-list and 3 if in the V-list.

The structure "red_result" shown in Figure 4.2 here is used to represent the active term and the current rule. The ALU-list uses the fields below. The field "isred" indicates if the term is reducible. The fields "rule" and "rulej" are the two indexes of rules. The field "class" is used in Smaran to store the class of the term. The field "Sig" is the pointer to a term.

```
Structure: red_result
    BOOL            isred;
    UINT            *rule;
    UINT            *rulej;
    UINT            class;
    struct signature *Sig;
    Red_Stack       *Stack;
    UINT            Top;
    struct SL_Node  *SL_Position;
```

Figure 4.2: Data structure of the red_result

Global variable "result" implemented by "red_result" indicates the active term and the current rule. Global variable "ALUresult" implemented by "red_result"

stores the first tuple in each step of normalization.

The structure "signature" has a field "val" implemented by the structure "Value" which stores the value of a term. The structure "Value" has a union "x" containing a field "VPtr" implemented by the structure of "variable". The ALU-list adds a field "instantiation" and a field "class" under the structure "variable" to store the instantiation information. Thus, the instantiation link for a term s is $s.val.x.VPtr.instantiation$ in TGR, and $s.val.x.VPtr.class$ in Smaran.

4.3 Integration into TGR

Normalization using the ALU-list in TGR is based on the ordinary TGR normalization functions. It also adds some new functions. The primary function in the ALU-list reduction strategy is "ALUnormaliseG" which is a variant of the function "normalise32G", the primary function in TGR. The essential new functions are listed below and the call graph of the below functions is shown in Figure 4.3.

- The non-recursive function "ALUnormaliseG" normalizes term t_0 to the normal form t_n if it exists. It is the main function with no input or output. It updates global variables representing the ALU-list, the DS-list, and the intermediate results if necessary.

- The non-recursive function "ALUinsertG" tries to build tuples for an instance of a point and insert them into the ALU-list. The first tuple of each step of normalization is kept out of the ALU-list. Its inputs are a pointer to the

candidate list, a pointer to the ALU-list, a pointer to the term, step number of normalization, and two integers for DE. It has no output. It updates the ALU-list and "ALUresult". MED can be used here to filter out tuples that cannot lead to any match.

- The non-recursive function "ALUinsert2G" tries to insert an instance of a point and its candidate list to the ALU-list recycled part. Details will be discussed in section 5.7.

- The non-recursive function "ALUpopG" pops a tuple (i, c, s) and tries to match s and lhs_c. If the match attempt fails, the function keeps popping until the ALU-list is empty or a successful match is found. The input is the ALU-list and the output is $true$ if a match is found, and $false$ otherwise.

- The non-recursive function "ALUnr_matchG" matches two terms and updates instantiation links if needed. The inputs are one LHS and a pointer to a term. The output is $true$ if match succeeds and $false$ otherwise.

- The non-recursive functions "ALUnr_reducible32G_DND" and "ALUDSL_reducibleG" look for a match for one DS s by browsing the group of rules the LHSs of which have same DS as s as the top symbol.

- The non-recursive function "ALUinsertVG" inserts the term into the V-list. The inputs are a pointer to the V-list, a pointer to the term. Details will be discussed in section 5.6.

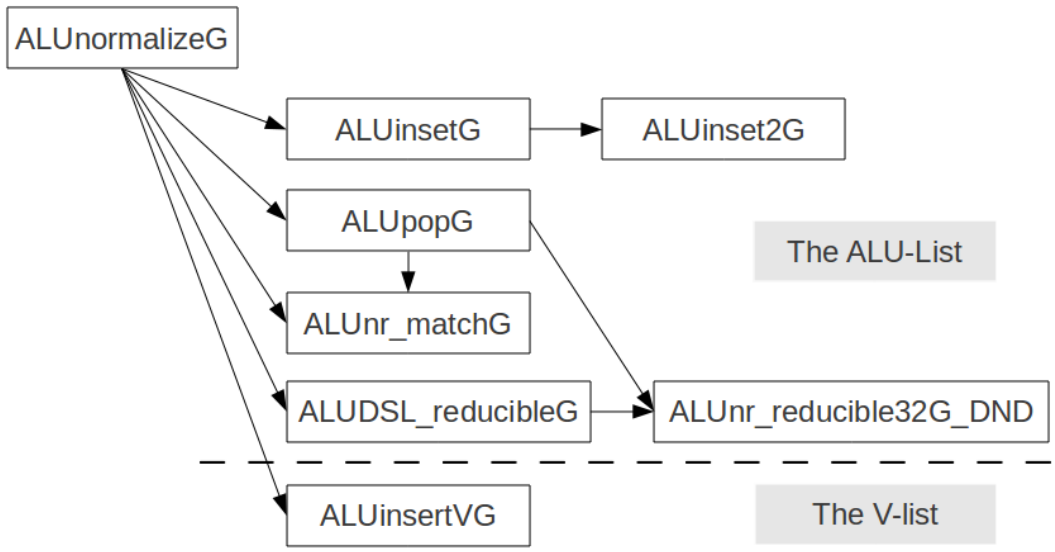

Figure 4.3: The call graph of the primary new functions in TGR with the ALU-list

4.3.1 The Function "ALUnormaliseG"

The main function "ALUnormaliseG" controls normalization. It initializes the ALU-list and calls TGR or the DS-list to find the active term and the current rule for the 1^{st} step before entering the main while loop which implements remaining steps of normalization. Each iteration of the while loop builds the instance of a RHS for the current step and looks for a match for the next step. When buiding the instance, the function follows three cases below.

1. If the RHS is a constant, the instance gets the constant.

2. If the RHS is a variable, the instance gets the instantiation. The function checks if the active term in current step was picked up by the ALU-list in the

previous step. If yes, the variable uses the instantiation link dedicated for the ALU-list. Otherwise the variable instantiation uses the original instantiation link.

3. If the RHS is a function, the functions starts from the bottom subterm to build the instance from the bottom up. Building the instance of each subterm follows these three cases. The function tries to evaluate the expression with predefined functions at the top. If the subterm is an ALU point, the function inserts a new pointer into the ALU-list.

After the current term is completed, the function tries to find the match for the next step. It attempts the first tuple. If the match succeeds, the function updates the active term and current rule for the next step. If not, the function goes to the ALU-list to find the match. If still no match is found and the ALU-list is empty, the function goes to other methods for the next match. Whenever a match is found, the function forwards to the next iteration. The function ends when it cannot find any match. Besides, the function updates information for the optimizations in Chapter 5. The pseudocode is below.

```
1: procedure ALUNORMALISEG
2:     initialize ALUlist
3:     initialize ALUMEVal                                  ▷ for MED
4:     loop = 0
5:     call ordinary method or the DS-list to find the match.
6:     update result.Sig, result.isred, result.rule, result.rulej
7:     while result.isred do
```

```
8:      loop = loop + 1
9:      currentrule = rule[result.rule][result.rulej]
10:     rhs = the RHS of currentrule
11:     switch rhs.Category do
12:         case function
13:             reset ALUMEVal                                  ▷ for MED
14:             r = bottom subterm of rhs
15:             while r ≠ rhs do
16:                 switch r.Category do
17:                     case constant
18:                         r.Sig = r.SPtr
19:                     case variable
20:                         if the match is found by the ALU-list then
21:                             r.Sig = r.val.x.VPtr.instantiation
22:                         else
23:                             r.Sig = r.Ptr.Sig
24:                         end if
25:                         if r.ALU_MEid then                  ▷ for MED
26:                             ALUMEVal[r.ALU_MEid] = r.Sig.val
27:                         end if
28:                         if the V-list is enabled AND r.Sig ∈ DS
29:                             AND r.Sig.ALU_DAV == 0 then     ▷ for the V-list
30:                             ALUInsertVG(ALUlist, r.Sig)
```

```
31:                     end if
32:                 case function
33:                     evaluate built-in operations
34:                     build the instance of r
35:                     result.Sig = r.Sig
36:                     if r.ALU_point == 1 then
37:                         ALUinsertG(r.ALU_candidate, ALUlist,
38:                         result.Sig, loop, r.ALU_dtime,
39:                         r.ALU_ftime)
40:                     end if
41:                     if r.ALU_MEid then                          ▷ for MED
42:                         ALUMEVal[r.ALU_MEid] = r.Sig.val
43:                     end if
44:             r = next subterm
45:         end while
46:     case constant
47:         result.Sig = r.SPtr
48:     case variable
49:         if the match is found by the ALU-list then
50:             result.Sig = r.val.x.VPtr.instantiation
51:         else
52:             result.Sig = r.Ptr.Sig
53:         end if
```

54: **if** the V-list is enabled AND $result.Sig \in DS$

55: AND $result.Sig.ALU\_DAV == 0$ **then** ▷ for the V-list

56: ALUINSERTVG($ALUlist, result.Sig$)

57: **end if**

▷ The current step of normalization ends here. Next step starts below

58: **if** ALUNR_MATCHG($ALUresult.LHSi, ALUresult.Sig$) **then**

59: ▷ Lazy Direct Match

60: $result.Sig = ALUresult.Sig$

61: $result.isred = true$

62: $result.rule = ALUresult.LHSi$

63: $result.rulej = ALUresult.LHSj$

64: **else**

65: insert the tuple $ALUresult$ into the ALU-list recycled part

66: ALUPOPG($ALUlist$)

67: **end if**

68: **if** match not found **then**

69: **if** $ALUlist.Size2 > 0$ **then** ▷ Recyclable ALU-list

70: $ALUlist.Size = ALUlist.Size2$

71: $ALUlist.Size2 = 0$

72: $ALUlist.Head = ALUlist.Fresh$

73: $ALUlist.Fresh = ALUlist.Tail$

74: **end if**

75: resort to other strategies

76: **end if**

77: **if** match not find **then**

78: $result.isred = false$

79: **end if**

80: **end while**

81: **end procedure**

The function calls original reduction method and strategy or the DS-list for the first step of normalization in line 5. The main while loop starts from line 7. It implements the three cases using the switch statement starting from line 11 when building the instance. The instantiation link dedicated for the ALU-list is implemented by $val.x.VPtr.instantiation$ while the original instantiation link is implemented by $Ptr.Sig$. The function builds the instance from the bottom up in line 14 and calls the function "ALUinsertG" to update the ALU-list whenever it visits an ALU point in line 37. To find the next match, firstly, it calls the function "ALUnr_matchG" to match the first tuple stored temporarily in "ALUresult" in line 58. Secondly, it calls the function"ALUpopG" in line 66. Thirdly, it routes to other strategies in line 75. Whenever a match is found, the function goes to the next iteration.

4.3.2 The Function "ALUinsertG"

The function inserts tuples into the ALU-list excluding the first tuple. This function is called whenever normalization visits an ALU point in a RHS. It traverses the candidate list and creates a tuple for each candidate. If MED is needed, it picks at

most one candidate. If MED is needed and it eliminates all candidates, the function inserts the instance and its candidate list to the ALU-list recycled part. Details will be discussed in section 5.2. The first tuple is stored temporarily and the remaining tuple are inserted into the ALU-list. The tuple is inserted after the node pointed by the field "Active". The ALU-list is managed as a stack. We need to make sure that a term can only be in one of the three lists, the DS-list, the ALU-list, and the V-list. We check before we insert any term into any list. Also, we make sure that after a term is in one list, it is not in any other list. The pseudocode is below.

```
1:  procedure ALUINSERTG(canlist, ALUlist, sig, loop, dtime, ftime)
2:      candidate = canlist.Head
3:      for i = 1 → canlist.size do
4:          flag = 1
5:          candidate = candidate.Next
6:          if MED is needed then                                  ▷ for MED
7:              MED checks candidate
8:              if MED eliminates candidate then
9:                  counter = counter + 1
10:                 flag = 0
11:             end if
12:         end if
13:         if flag == 0 AND counter ≥ canlist.size  then
14:             ALUINSERT2G(canlist, sig, loop, min, DF, max)
15:         end if
```

16: **if** $flag == 1$ **then**

17: **if** *candidate* is the 1^{st} candidate **then**

18: $ALUresult.Sig = sig$

19: $ALUresult.rule = candidate.LHSi$

20: $ALUresult.rulej = candidate.LHSj$

21: **else**

22: initialize a tuple *tuple*

23: $tuple.Sig = sig$

24: $tuple.LHSi = candidate.LHSi$

25: $tuple.LHSj = candidate.LHSj$

26: $tuple.Same = candidate.Same$ ▷ for SPE

27: $tuple.DFMin = dtime$ ▷ for DE

28: $tuple.DFMax = ftime$ ▷ for DE

29: $tuple.Next = NULL$

30: add *tuple* after $ALUlist.Active$

31: $sig.ALU\_DAV = 2$

32: $ALUlist.Size = ALUlist.Size + 1$

33: update $ALUlist.Active$, $ALUlist.Fresh$, $ALUlist.Tail$ if needed

34: **if** *sig* is in the DS-list **then**

35: delete *sig* from the DS-list

36: **end if**

37: **end if**

38: **if** MED is needed **then**

```
39:                 break
40:             end if
41:         end if
42:     end for
43: end procedure
```

The function uses a main for loop to traverse the candidate list starting from line 3. It stores the first tuple for each step temporarily in the global variable "ALUresult" in line 17. The function builds the tuple starting from line 22. The field "ALU_DAV" is used in line 31 to mark that the subterm is in the ALU-list. The non-recursive function "ALUinsert2G" in line 14 adds the instance of a point and its candidate list to the end of the ALU-list recycled part. Details will be discussed in section 5.7.

4.3.3 The Function "ALUpopG"

The function tries to pick the active term and the current rule from the ALU-list. It pops the top node (i, c, s) and attempts to match the term represented by s with lhs_c. Upon a successful match, the function updates the active term and the current rule for the next step, eliminates candidates using DE and SPE, and ends. If no match is found, the function pops the next node. The function also ends when the ALU-list fresh part is empty and no match is found. If the V-list is enabled, the function continues to pop nodes in the V-list if no match is found in the ALU-list fresh part, and ends when either a match is found or the V-list is empty. The pseudocode is below.

```
1: procedure ALUPOPG(ALUlist)
2:     while ALUlist.Size > 0 AND no match is found do
3:         ALUlist.Head = ALUlist.Head.Next
4:         ALUlist.Size = ALUlist.Size − 1
5:         ALUlist.Head.Sig.ALU_DAV = 0
6:         update ALUlist.Active, ALUlist.Fresh, ALUlist.Tail if needed
7:         if ALUNR_MATCHG(ALUlist.Head.LHSi, ALUlist.Head.Sig) then
8:             result.Sig = ALUlist.Head.Sig
9:             result.isred = true
10:            result.rule = ALUlist.Head.LHSi
11:            result.rulej = ALUlist.Head.LHSj
12:        else
13:            insert the tuple ALUlist.Head into the ALU-list recycled part
14:        end if
15:        if match is found AND ALUlist.Size > 0 then
16:            temp = ALUlist.Head.Next                          ▷ for SPE
17:            same = ALUlist.Head.Same
18:            loop = ALUlist.Head.Loop
19:            while temp.Same == same AND temp.Loop == loop
20:                AND temp ≠ NULL do
21:                temp.Sig.ALU_DAV = 0
22:                ALUlist.Size = ALUlist.Size − 1
23:                ALUlist.Head = ALUlist.Head.Next
```

24: $temp = ALUlist.Head.Next$

25: **end while**

26: Descendant Elimination

27: **end if**

28: **end while**

29: **if** the V-list is enabled AND no match is found **then** ▷ for the V-list

30: **while** $ALUlist.VSize > 0$ AND no match is found **do**

31: pop the top tuple to look for a match

32: **end while**

33: **end if**

34: **if** match is found **then**

35: **return** $true$

36: **else**

37: **return** $false$

38: **end if**

39: **end procedure**

The function consists of two while loops. The first one starting from line 2 goes around the ALU-list until a successful match is found or the list is empty. It pops the top node and calls the function "ALUnr_matchG" to match in line 7. The function updates the active term and the current rule for the next step starting from line 8. The second while loop runs only when the V-list in section 5.6 is enabled.

4.3.4 The Function "ALUnr_matchG"

The function uses a local stack to DFS an LHS and a term at the same time. If the subterm in the LHS is a variable, the function updates its instantiation link. If not, the function compares the subterm of the term and the subterm of the LHS syntactically. It returns true if match succeeds. The pseudocode is below.

```
1:  procedure ALUnr_matchG(lhsi, sig)
2:      lhs = the LHS of the rule indexed by lhsi and the intrep of sig
3:      initialize ALUDE                                          ▷ for DE
4:      while true do
5:          for i = 1 → sig.arity do
6:              l = lhs.parameters[i]
7:              s = sig.parameters[i]
8:              switch s.Category, l.Category do
9:                  case constant, constant
10:                     if s ≠ l then
11:                         return false
12:                     end if
13:                 case constant, variable
14:                     l.val.x.VPtr.instantiation = s
15:                 case constant, default
16:                     return false
17:                 case function, function
18:                     if root(s) == root(l)  then
```

```
19:                     push l and s into the stack
20:                 end if
21:             case function, variable
22:                 l.val.x.VPtr.instantiation = s
23:                 ALUDECnt = ALUDECnt + 1                    ▷ for DE
24:                 ALUDE[ALUDECnt * 2] = s.ALU_dtime
25:                 ALUDE[ALUDECnt * 2 + 1] = s.ALU_ftime
26:             case function, default
27:                 return false
28:         end for
29:         if stack is not empty then
30:             pop lhs and sig from the stack
31:         else
32:             break
33:         end if
34:     end while
35:     return true
36: end procedure
```

The function consists of a while loop starting from line 4 to traverse both the input term and input LHS. It compares the subterm according to their categories in the switch statement starting from 8. DE collects information when a variable instantiates to a function in line 23.

4.4 Integration into Smaran

Integration into Smaran is similar to the integration into TGR. The main difference is that, to represent a term, Smaran uses a class and its unreduced signature while TGR uses the term itself. Thus, in Smaran, normalization actually stores the class number in s in the tuple (i, c, s) and tries to match between the unreduced signature of the class in s and lhs_c. Recall TGR stores the pointer to the term in s and match happens between the term and lhs_c.

Smaran uses a global array "xClass_List" implemented in the structure "xClass" to store the classes of intermediate results. The normal form if it exists is represented by the unreduced signature of a class c, $xClass\_list[c].unred$.

Normalization with the ALU-list with Smaran follows regular procedure. The ALU-list tries to find the active term and the current rule except for the first step. If the ALU-list cannot find them, normalization goes back to other methods. Then normalization builds the instance and updates the ALU-list. The primary function is "ALUnormailse" which calls some new functions. The primary new functions are basically the variants of the new functions in section 4.3 and the call graph is shown in Figure 4.4.

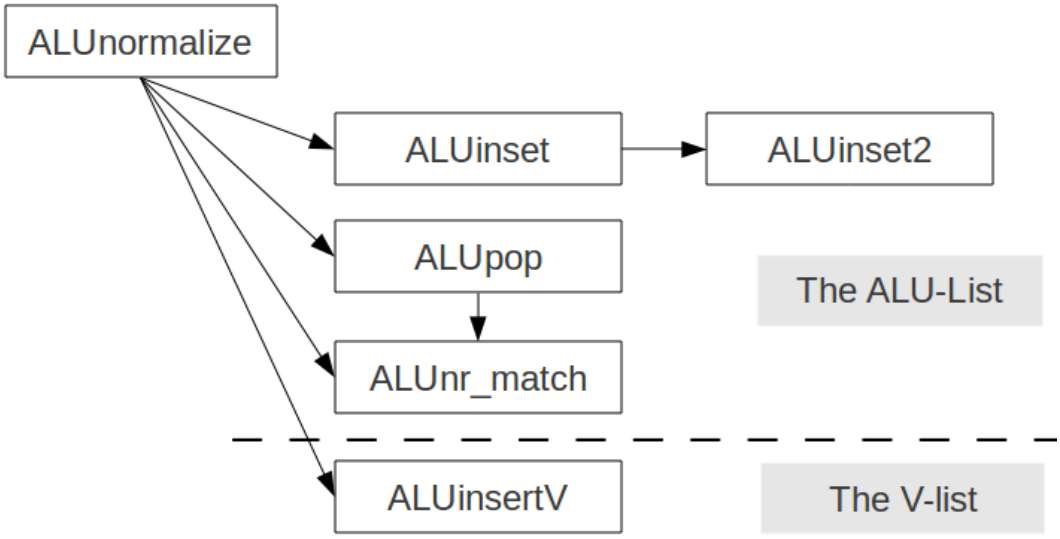

Figure 4.4: The call graph of the primary new functions in Smaran with the ALU-list

4.4.1 The Function "ALUnormalise"

The non-recursive function implements normalization with Smaran. The algorithm is as same as the algorithm of the function "ALUnormaliseG". In a regular loop, it builds the instance, updates the ALU-list, looks for the active term and current rule, and goes to the next step. The function has no input or output, and ends with a normal form if it exists. The pseudocode is below.

1: **procedure** ALUNORMALISE
2: initialize $ALUlist$
3: initialize $ALUMEVal$ ▷ for MED
4: $loop = 0$
5: call ordinary method or the DS-list to find the match.

```
6:  updates result.Sig, result.isred, result.rule, result.rulej
7:  while result.isred do
8:      loop = loop + 1
9:      currentrule = rule[result.rule][result.rulej]
10:     rhs = the RHS of currentrule
11:     switch rhs.Category do
12:         case function
13:             r = bottom subterm of rhs
14:             reset ALUMEVal                                  ▷ for MED
15:             while r ≠ rhs do
16:                 switch r.Category do
17:                     case constant
18:                         r.Class = class of r.SPtr          ▷ different from TGR
19:                     case variable
20:                         if the match is found by the ALU-list then
21:                             r.Class = r.val.x.VPtr.class   ▷ different from TGR
22:                         else
23:                             r.Class = r.Ptr.Class          ▷ different from TGR
24:                         end if
25:                         if r.ALU_MEid then                 ▷ for MED
26:                             ALUMEVal[r.ALU_MEid]
27:                             = xClass_List[r.Class].Unred.val
28:                         end if
```

29: **if** the V-list is enabled ▷ for the V-list

30: AND $xClass\_List[r.class].Unred \in DS$

31: AND $xClass\_List[r.class].ALU\_DAV == 0$ **then**

32: ALUINSERTV($ALUlist, r.Class$)

33: **end if**

34: **case** function

35: expression evaluation

36: build the instance of r

37: $result.class = r.Class$

38: **if** $r.ALU\_point == 1$ **then**

39: ALUINSERT($r.ALU\_candidate, ALUlist,$

40: $result.Sig, loop, r.ALU\_dtime,$

41: $r.ALU\_ftime$) ▷ different from TGR

42: **end if**

43: **if** $r.ALU\_MEid$ **then** ▷ for MED

44: $ALUMEVal[r.ALU\_MEid]$

45: $= xClass\_List[r.Class].Unred.val$

46: **end if**

47: r = next subterm

48: **end while**

49: **case** constant

50: $result.Class$ = class of $r.SPtr$ ▷ different from TGR

51: **case** variable

52: **if** the match is found by the ALU-list **then**

53: $result.class = r.val.x.VPtr.class$ ▷ different from TGR

54: **else**

55: $result.class = r.Ptr.Class$ ▷ different from TGR

56: **end if**

57: **if** the V-list is enabled ▷ for the V-list

58: AND $xClass\_List[result.class].Unred \in DS$

59: AND $xClass\_List[result.class].ALU\_DAV == 0$ **then**

60: ALUINSERTV$(ALUlist, result.Class)$

61: **end if**

▷ The current step of normalization ends. Next step starts

62: **if** ALUNR_MATCH$(ALUresult.LHSi,$ ▷ Lazy Direct Match

63: $xClass\_List[ALUresult.class].Unred)$ **then** ▷ different from TGR

64: $result.class = ALUresult.class$ ▷ different from TGR

65: $result.isred = true$

66: $result.rule = ALUresult.LHSi$

67: $result.rulej = ALUresult.LHSj$

68: **else**

69: insert the tuple $ALUresult$ into the ALU-list recycled part

70: ALUPOP$(ALUlist)$ ▷ different from TGR

71: **end if**

72: **if** match not found **then**

73: resort to other strategies

```
74:          if ALUlist.Size2 > 0 then                    ▷ Recyclable ALU-list
75:              ALUlist.Size = ALUlist.Size2
76:              ALUlist.Size2 = 0
77:              ALUlist.Head = ALUlist.Fresh
78:              ALUlist.Fresh = ALUlist.Tail
79:          end if
80:      end if
81:      if match not find then
82:          result.isred = false
83:      end if
84:  end while
85: end procedure
```

4.4.2 The Function "ALUinsert"

The non-recursive function builds a tuple (i, c, s) for each unification pair. It only needs the class number for s. The CSD requires the term additionally. The function stores the first tuple in every step temporarily in the global variable "ALUresult". The remaining tuples are added into the ALU-list. The input is same as the input of the function "ALUinsertG". So is the output. The algorithm is as same as the algorithm of the function "ALUinsertG". The pseudocode is below.

```
1: procedure ALUINSERT(canlist, ALUlist, sig, loop, dtime, ftime)
2:     candidate = canlist.Head
3:     class = class of sig                              ▷ no need in TGR
```

```
4:  for i = 1 → canlist.size do
5:      flag = 1
6:      candidate = candidate.Next
7:      if MED is needed then
8:          MED checks candidate
9:          if MED eliminates candidate then
10:             counter = counter + 1
11:             flag = 0
12:         end if
13:     end if
14:     if flag == 0 AND counter ≥ canlist.size then
15:         ALUINSERT2(canlist, sig, loop, min, max)
16:     end if
17:     if flag == 1 then
18:         if candidate is the 1st candidate then
19:             ALUresult.class = class                ▷ different from TGR
20:             ALUresult.rule = candidate.LHSi
21:             ALUresult.rulej = candidate.LHSj
22:         else
23:             initialize a tuple tuple
24:             tuple.Class = class                    ▷ different from TGR
25:             tuple.Sig = sig                        ▷ for CSD
26:             tuple.LHSi = candidate.LHSi
```

27: $tuple.LHSj = candidate.LHSj$

28: $tuple.Same = candidate.Same$ ▷ for SPE

29: $tuple.DFMin = dtime$ ▷ for DE

30: $tuple.DFMax = ftime$ ▷ for DE

31: $tuple.Next = NULL$

32: add the *tuple* after $ALUlist.Active$

33: $xClass\_List[class].ALU\_DAV = 2$ ▷ different from TGR

34: $ALUlist.Size = ALUlist.Size + 1$

35: update $ALUlist.Active$, $ALUlist.Fresh$, $ALUlist.Tail$ if needed

36: **if** *sig* is in the DS-list **then**

37: delete *class* from the DS-list ▷ different from TGR

38: **end if**

39: **end if**

40: **if** MED is needed **then**

41: break

42: **end if**

43: **end if**

44: **end for**

45: **end procedure**

The non-recursive function "ALUinsert2" in line 15 is a variant of "ALUinsert2G". It stores the class number of the term. Details will be discussed in section 5.7.

4.4.3 The Function "ALUpop"

The non-recursive "ALUpop" pops a tuple (i, c, s) and tries to match the unreduced signature of s and lhs_c which is the only difference from the function"ALUpopG". The pseudocode is below.

```
1: procedure ALUPOP(ALUlist)
2:     while ALUlist.Size > 0 AND no match is found do
3:         ALUlist.Head = ALUlist.Head.Next
4:         ALUlist.Size = ALUlist.Size − 1
5:         class = ALUlist.Head.Class                    ▷ no need in TGR
6:         xClass_List[class].ALU_DAV = 0                ▷ different from TGR
7:         update ALUlist.Active, ALUlist.Fresh, ALUlist.Tail if needed
8:         if xClass_List[class].Unred == ALUlist.Head.Sig then      ▷ for CSD
9:             if ALUNR_MATCH(ALUlist.Head.LHSi,
10:                xClass_List[class].Unred) then         ▷ different from TGR
11:                result.class = class                   ▷ different from TGR
12:                result.isred = true
13:                result.rule = ALUlist.Head.LHSi
14:                result.rulej = ALUlist.Head.LHSj
15:            else
16:                insert the tuple ALUlist.Head into the ALU-list recycled part
17:            end if
18:        end if
19:        if match is found AND ALUlist.Size > 0 then
```

20: $temp = ALUlist.Head.Next$ ▷ for SPE

21: $same = ALUlist.Head.Same$

22: $loop = ALUlist.Head.Loop$

23: **while** $temp.Same == same$ AND $temp.Loop == loop$

24: AND $temp \neq NULL$ **do**

25: $xClass\_List[temp.Class].ALU\_DAV = 0$ ▷ different from TGR

26: $ALUlist.Size = ALUlist.Size - 1$

27: $ALUlist.Head = ALUlist.Head.Next$

28: $temp = ALUlist.Head.Next$

29: **end while**

30: Descendant Elimination

31: **end if**

32: **end while**

33: **if** the V-list is enabled AND no match is found **then** ▷ for the V-list

34: **while** $ALUlist.VSize > 0$ AND no match is found **do**

35: pop the top tuple to look for a match

36: **end while**

37: **end if**

38: **if** match is found **then**

39: **return** 1

40: **else**

41: **return** 0

42: **end if**

43: **end procedure**

The non-recursive function "ALUinsertV" inserts the class number not the term into the V-list.

4.4.4 The Function "ALUnr_match"

The function "ALUnr_match" is close to "ALUnr_matchG". The difference is that "ALUnr_match" instantiates to a class not a subterm. It returns true if match succeeds. The pseudocode is below.

```
1:  procedure ALUNR_MATCH(lhsi, sig)
2:      lhs = the LHS of the rule indexed by lhsi and the intrep of sig
3:      initialize ALUDE                                          ▷ for DE
4:      while true do
5:          for i = 1 → sig.arity do
6:              l = lhs.parameters[i]
7:              s = the unreduced signature of sig.parameters[i]
8:              class = the class of s
9:              switch s.Category, l.Category do
10:                 case constant, constant
11:                     if s ≠ l then
12:                         return false
13:                     end if
14:                 case constant, variable
```

15: $l.val.x.VPtr.instantiation = class$ ▷ different from TGR

16: **case** constant, default

17: **return** $false$

18: **case** function, function

19: **if** $root(s) == root(l)$ **then**

20: push l and s into the stack

21: **end if**

22: **case** function, variable

23: $l.val.x.VPtr.instantiation = class$ ▷ different from TGR

24: $ALUDECnt = ALUDECnt + 1$ ▷ for DE

25: $ALUDE[ALUDECnt * 2] = s.ALU\_dtime$

26: $ALUDE[ALUDECnt * 2 + 1] = s.ALU\_ftime$

27: **case** function, default

28: **return** $false$

29: **end for**

30: **if** stack is not empty **then**

31: pop lhs and sig from the stack

32: **else**

33: break

34: **end if**

35: **end while**

36: **return** $true$

37: **end procedure**

UP thoroughly collects the unification results which are effectively integrated into normalization with both TGR and Smaran. We find that the unification helps normalization. Additionally, we will introduce some optimizations which make normalization more accurate and faster.

Chapter 5

Optimizations

This chapter will discuss the optimizations we implement to improve the efficiency of LRR. Optimizations on the ALU-list include fast prediction in section 5.1, Mutually Exclusive Detection in section 5.2, Descendant Elimination in section 5.3, Same Point Elimination in section 5.4, Changed Signature Detection in section 5.5, the V-list in section 5.6, and the Recyclable ALU-list in section 5.7. Section 5.8 presents optimizations on the memory management. Optimizations on the DS-list and statistics are explained in section 5.9 and 5.10.

5.1 Fast Prediction

Fast prediction reduces the overhead when the ALU-list produces a normalization tuple (i, c, s) from a unification pair (C, P) and when the ALU-list predicts the match. Fast prediction contains two parts, elimination of path lists and direct match.

5.1.1 Elimination of Path Lists

Previous UP stored the path in a list for each ALU point before normalization. And in normalization, after the instance completed, previous ALU-list followed the path from the root of the current term to look for the instance of the ALU point. We eliminate path lists to save the space and to save the time in locating s in the tuple (i, c, s).

As we discussed in section 3.2, the latest UP implements conceptual unification pair (C, P) by flagging the ALU point at location P and storing a list of candidates C. The field "ALU_point" in the data structure "RHS" in Figure 3.3 flags the point and the field "ALU_candidate" leads the candidate list. The field "ALU_point" helps to locate the instance of an ALU point faster without extra traversal in normalization. When building the instance of a RHS in normalization, LRR traverses the RHS from the bottom up. When LRR visits the ALU point (the value of its "ALU_point" is equal to 1) in the RHS, it builds its instance, s, which is needed by the ALU-list. Thus, the ALU-list keeps s. TGR stores the pointer to s in the function "ALUnormaliseG" while Smaran stores the class of s in the function "ALUnormalise". Please see line 36 in "ALUnormailiseG" and line 38 in the function "ALUnormailse". We just need one full traversal of a RHS to both build the instance and collect the instances of all ALU points.

5.1.2 Direct Match

In each step of normalization, previous LRR used to call the function "ALUinsertG" or "ALUinsert" to add every tuple into the ALU-list and call the function "ALUpopG" or "ALUpop" to pop the top one for a match. We notice that the first tuple found in each step actually matches with great accuracy especially with the help of MED. Thus, to save time in function calls, current LRR stores the first tuple temporarily in a global variable "ALUresult" (Please see line 17 in "ALUinsertG" and line 18 in the function "ALUinsert") and directly matches the tuple by calling the function "ALUnr_matchG" or "ALUnr_match". If there are more than one tuple in one step, rest tuples still need to be inserted into the ALU-list. If the first tuple matches, no popping is needed. Otherwise, normalization puts the first tuple into the ALU recycled part and pops the top tuple from the ALU-list. To compute Fibonacci of 23 using TGR and the ALU-list, LRR calls the function "ALUnr_matchG" 185,469 times, only 46,367 times (25.00%) is called by the function "ALUpopG". We save 75.00% calls to both the function "ALUinsertG" and "ALUpopG".

There are two ways to implement direct match, lazy direct match and eager direct match. Lazy direct match waits until the intermediate result completes to match the first tuple. This happens at the beginning of the following step. Please see line 58 in "ALUnormailiseG" and line 62 in "ALUnormailse". Eager direct match tries to match immediately after the first tuple is created. In current step, it looks for a match for the next step. We prefer lazy direct match because it calls function "ALUnr_matchG" or "ALUnr_match" which updates instantiation links for the next step besides matching. On the other hand, since eager direct match happens when

the instance is incomplete, it has to either keep instantiation links for the current step and instantiation links for the next step separately or keep matching and updating instantiations separately.

For example, we have a RHS $f(x, g(x))$ in which $g(x)$ unifies with an LHS, l. In either TGR or Smaran, the instantiation of x is shared. Normalization builds its instance from the bottom up. In the i^{th} step, x is instantiated to 2. $(i, \sigma(g(x)), l)$ is the first tuple. Lazy direct match waits until the $i + 1^{th}$ step to match $\sigma(g(x))$ and x is instantiated to 4. Eager direct match tries to match in the i^{th} step. However, it cannot update current instantiation of x to 4 because another x (the first parameter of the function f) should be instantiated to 2 as well. Thus, eager direct match has to either use a separate link to store the instantiation 4 for the $i + 1^{th}$ step or purely match in the i^{th} step and purly update instantiation links in the $i + 1^{th}$ step.

5.2 Mutually Exclusive Detection

MED cuts unnecessary insertions into the ALU-list caused by *Mutually Exclusive (ME) subterms* in the LHSs. Most ME subterms are introduced by the extension of the ALU algorithm. Before normalization, MED detects ME subterms from the rule set R. Then during normalization, it eliminates tuples that will never match successfully.

Initially, we brought in MED due to the fact that the extension of the ALU algorithm considers every possible result of a predefined operation as a candidate in UP and in normalization only one candidate may succeed in matching. For example, back

to the Fibonacci calculator in section 1.3. Both $lhs_2, f(true, x)$ and $lhs_3, f(false, x)$ are candidates for the ALU point $rhs_1, f(>(x, 1), x)$. $true$ and $false$ are called ME subterms, and $>(x, 1)$ is located at an *ME point*. In one step of normalization, if $x = 2$, lhs_2 will succeed in matching while lhs_3 will definitely fail. MED prevents the tuple containing lhs_3 from insertion into the ALU-list or from direct match. The latest MED prevents every tuple that contains a ME subterm in the candidate and will fail in matching.

MED contains three parts: collecting ME subterms, labeling ME points, eliminating tuples. MED collects ME subterms at the beginning of UP so that during UP, MED can label the ME points. In normalization, MED eliminates unnecessary tuples according to ME terms and ME points.

5.2.1 Collecting ME Subterms

ME subterms locate at the same location of a group of different LHSs which have the same DS at the root. At the beginning of UP before normalization, MED collects ME subterms by unifying every pair of LHSs that have the same DS at the root. Subterms that cause a pair of LHSs *not* to unify are defined as ME subterms. For example, in the Fibonacci calculator, MED unifies lhs_2 and lhs_3. They don't unify because $true$ is not unifiable with $false$. $true$ and $false$ are ME subterms.

We reuse the existing structure "Value" to store a ME subterm. We then create a global array ALUME to store all ME subterms, and a global integer "ALUMECnt" to be the counter of the array. The primary functions are "ALUME_read" and

"ALUunify_LHS". Figure 5.1 shows the data structure of the structure "Value". According to the field "Type", one field under the union "x" stores the value. Figure 5.2 shows the call graph of ME subterm collection.

```
Structure: Value
   USHORT                Type;
   Union:          x
      INT                Integer;
      REAL               Float;
      USHORT             Char;
      USHORT             *String;
      struct Set         *set;
      struct variable    *VPtr;
      struct constant    *CPtr;
      struct list        *FPtr;
      struct signature   *Sig;
```

Figure 5.1: The structure "Value"

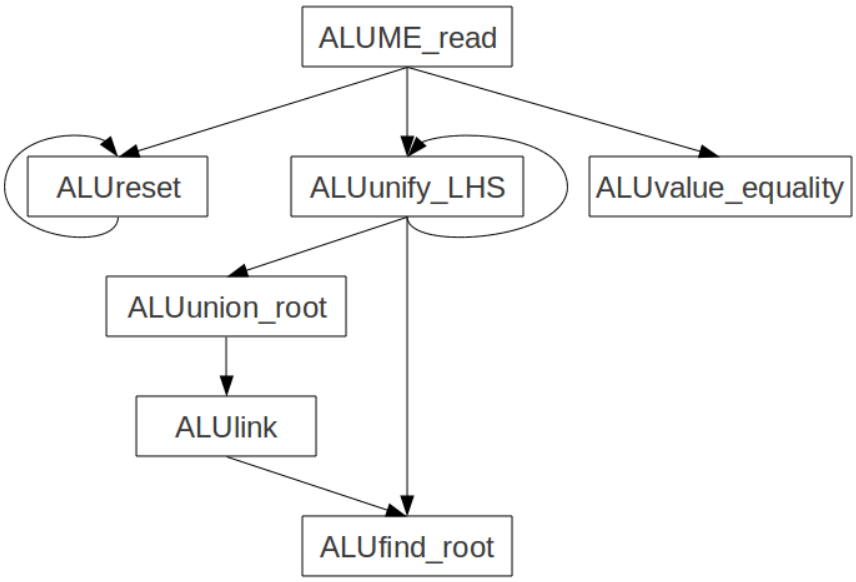

Figure 5.2: The call graph of primary functions in collecting ME subterms

The function "ALUME_read" unifies every pair of LHSs that have same top DS. It consists of a main for loop to traverse every pair of these LHSs. For each pair, it tries to unify and then resets two LHSs. Since we collect ME subterms from failed unifications, there is no need to check any loop. The pseudocode is below.

1: **procedure** ALUME_READ
2: initialize the array $ALUME$
3: **for** each pair of l_1 and l_2 that have same root symbol **do**
4: ALUUNIFY_LHS(l_1, l_2)
5: ALURESET(l_1)
6: ALURESET(l_2)
7: **end for**
8: **end procedure**

The function calls the function "ALUunify_LHS" to unify one pair of LHSs and the function "ALUreset" to reset an LHS after unification.

The function "ALUunify_LHS" is similar to the function "ALUunify". The difference is that before the unification fails, "ALUunify_LHS" inserts the two ME subterms into the global array "ALUME" if they are not in the array. The pseudocode is below.

1: **procedure** ALUUNIFY_LHS(t_1, t_2)
2: s_1 = ALUFIND_ROOT(t_1)
3: s_2 = ALUFIND_ROOT(t_2)
4: **if** $s_1 == s_2$ **then return** $true$
5: **end if**

```
6:    switch s1.Category, s2.Category do
7:        case variable, variable
8:            ALUunion_root(s1, s2)
9:            return true
10:       case variable, non-variable
11:           s1.ALU_is = s2
12:           return true
13:       case non-variable, variable
14:           s2.ALU_is = s1
15:           return true
16:       case default
17:           if s1 and s2 have same function symbol then
18:               ALUunion_root(s1, s2)
19:               for i = 1 → s1.arity do
20:                   if ALUunify_LHS(s1.ALU_para[i], s2.ALU_para[i])
21:                       == false then
22:                       return false
23:                   end if
24:               end for
25:               return true
26:           else
      ▷ Collection of ME subterms starts.
27:               mefound = false
```

```
28:     for j = 1 → ALUMECnt do
29:         if ALUvalue_equality(s1.val, ALUME[j])
30:             == true then
31:             mefound = true
32:             break
33:         end if
34:     end for
35:     if mefound == false then
36:         ALUMECnt = ALUMECnt + 1
37:         ALUME[ALUMECnt] = s1.val
38:     end if
39:     mefound = false
40:     for j = 1 → ALUMECnt do
41:         if ALUvalue_equality(s2.val, ALUME[j]) then
42:             mefound = true
43:             break
44:         end if
45:     end for
46:     if mefound == false then
47:         ALUMECnt = ALUMECnt + 1
48:         ALUME[ALUMECnt] = s2.val
49:     end if
50:     return false
```

51: **end if**

52: **end procedure**

The function calls functions "ALUfind_root", "ALUunion_root" for unification. It calls the function "ALUvalue_equality" to compare the values of two subterms.

5.2.2 Labeling ME Points

Similar to ALU points, ME points are the positions of subterms in RHSs that unify with ME subterms in LHSs. ME subterms are subterms in LHSs and ME points are locations in RHSs. When UP unifies every subterm in every RHS with every LHS, it also labels every subterm that unifies with an ME subterm. Every subterm t located at an ALU point in the RHS unifies with an LHS. Thus, $root(t) \in DS$. Every subterm s located at an ME point in the RHS unifies with an ME subterm in an LHS. Thus, $root(s) \in predefined\ operators$ and a subterm at an ME point is a descendant of another subterm at an ALU point. For each RHS, we label each ME point with a unique id, defined as *ME ID*. We also use *ME TOTAL* to count how many ME points are there for each RHS and *ME MAXID* to record the maximum ME ID for each ALU point. A RHS r with *total* ME points may have multiple ALU points and each ALU point may have multiple candidates. To each candidate C that unifies with a subterm under r, we associate a ME array with *total* elements. Every element is initialized to empty. For each ME subterm under C that unifies with a subterm at an ME point with ME ID id, we store the ME subterm as the id^{th} element in the ME array associated with C. Since usually an ALU point of r cannot

cover every ME point of r, a candidate under r cannot cover every ME subterm that unifies with every ME point of r. In most cases, ME array is not dense. For example, in the Fibonacci calculator, $true$ in lhs_2, and $false$ in lhs_3 are ME subterms. When UP is unifying rhs_1 with lhs_2, $>(x,1)$ unifies with $true$ in lhs_2. UP labels $>(x,1)$ as an ME point with ME ID 1 or the 1^{st} ME point. When UP successfully unifies rhs_1 with lhs_2, it builds a candidate lhs_2 with an ME Array storing $true$ as the 1^{st} element. Then UP unifies rhs_1 with lhs_3 and builds another candidate lhs_3 with an ME array storing $false$ as the 1^{st} element. A RHS may have multiple ME points, such as $f(>(x,1), g(>(y,2)), >(z,3))$ in which $f(>(x,1), g(>(y,2)), >(z,3))$ and $g(>(y,2))$ are both ALU points. Say $>(x,1)$ is the 1^{st} ME point. $>(y,2)$ is the 2^{nd} ME point. $>(z,3)$ is the 3^{rd} ME point. For a certain rule set, ALU point $g(>(y,2))$ has a candidate C_j with an ME array storing $true$ as the 2^{nd} element, $C_{j'}$ with an ME array storing $false$ as the 2^{nd} element. ALU point $f(>(x,1), g(>(y,2)), >(z,3))$ has a candidate C_i with an ME array storing $true$ as the 1^{st} element and $false$ as the 3^{rd} element and $C_{i'}$ with an ME array storing $false$ as the 1^{st} element and $false$ as the 3^{rd} element .

In the structure "RHS" in Figure 3.3, the field "ALU_MEid" is initialized to 0 and stores the ME ID. The field "ALU_MEcnt" stores ME TOTAL. In the structure "ALU_List_Node" in Figure 3.5, the field "MEVal" starts an ME array for each candidate. The field "MEValMax" stores ME MAXID. For example, Figure 5.3 shows $f(>(x,1), g(>(y,2)), >(z,3))$ above.

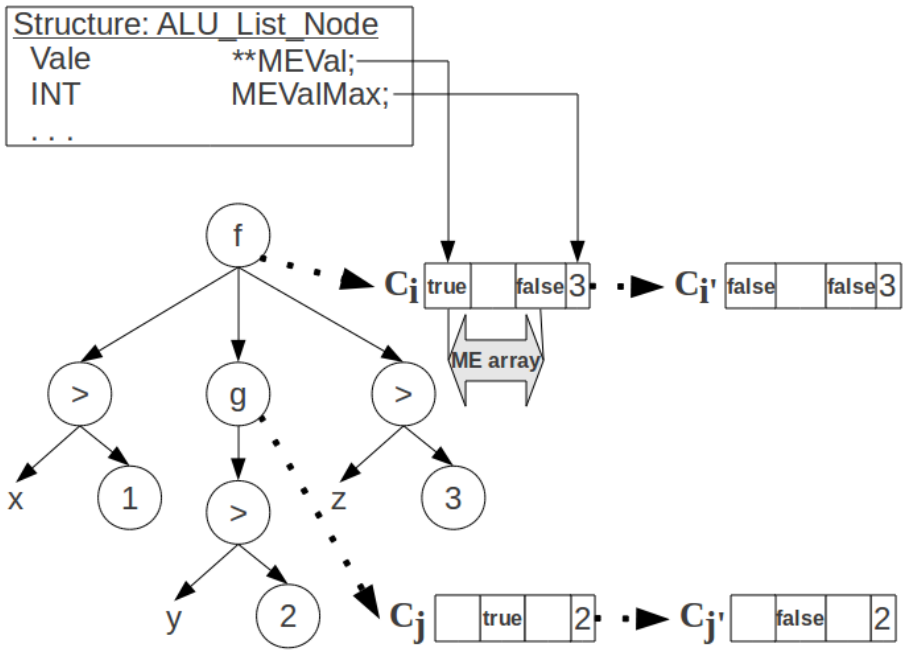

Figure 5.3: An example of ME terms and ME points

In the function "ALUunify", MED detects ME points in two cases: a) when the subterm in the LHS is not a variable and the subterm in the RHS is a variable; b) when the subterm in the LHS is *true* or *false* and the subterm in the RHS is a built-in Boolean operations. If the subterm in the LHS is a ME term, the subterm in the RHS is at an ME point. And then MED updates the ME array as necessary. In the function "ALUDFLM", after a successful unification, MED updates the field "MEVal" associating the ME array with a candidate and updates the field "MEValMax". Recall that in section 3.2.2.1, the field "ALU_para" in the structure "RHS" implements a separate unification dedicated DAG for a term. The function "ALUunify" updates the field "ALU_MEid" of a subterm implemented by "ALU_para" while normalization traverses subterms implemented by "parameters".

Thus, in the function "ALUDFLM" after all unifications complete, MED calls the function "ALUME_copy" to copy the value of field "ALU_MEid" from every subterm implemented by "ALU_para" to the corresponding subterm implemented by "parameters".

5.2.3 Eliminating Tuples

In one step of normalization, when LRR is building the instance of a RHS, MED builds a global ME array to store values of instances of all ME points. By comparing the global ME array with the local ME array of each candidate, MED eliminates unnecessary tuples. LRR traverses the RHS to build its instance. When LRR visits an ME point with ME ID id, MED puts the value of the instance into the id^{th} element of a global ME array. Since LRR builds the instance from the bottom up, when it visits an ALU point and tries to create a normalization tuple, all ME points under the ALU point have been visited. Thus, the values of the instances of the ME points are in the global array. MED compares the local ME array associated to the candidate with the global ME array. If the non-empty values in the local ME array does not match the peer values in the global ME array, MED eliminates the normalization tuples containing the candidate.

Take $f(>(x,1), g(>(y,2)), >(z,3))$ in Figure 5.3 as an example. Say in the i^{th} step, $x = 0$, $y = 1$, $z = 2$. When LRR builds the instance, $>(y,2)$ evaluates to $false$. The 2^{nd} element in the global ME array stores $false$. When normalization visits g, it builds two normalization tuples $(i, C_j, \sigma(g(false)))$ and $(i, C_{j'}, \sigma(g(false)))$.

By comparing the value of the 2^{nd} element between the local ME arrays and the global ME array, MED eliminates $(i, C_j, \sigma(g(false)))$, because the 2^{nd} element in its local ME array is $true$. Similarly, the 1^{st} element of the global ME array is $false$ and the 3^{rd} element of the global ME array is $false$. For the ALU point $f(>(x,1), g(>(y,2)), >(z,3))$, MED eliminates the tuple containing candidate C_i and keeps the tuple containing $C_{i'}$. Please see the results after MED in Figure 5.4. LRR will try to match the instance of $f(>(x,1), g(>(y,2)), >(z,3))$ with $lhs_{C_{i'}}$ and match the instance of $g(>(y,2))$ with $lhs_{C_{j'}}$ in future.

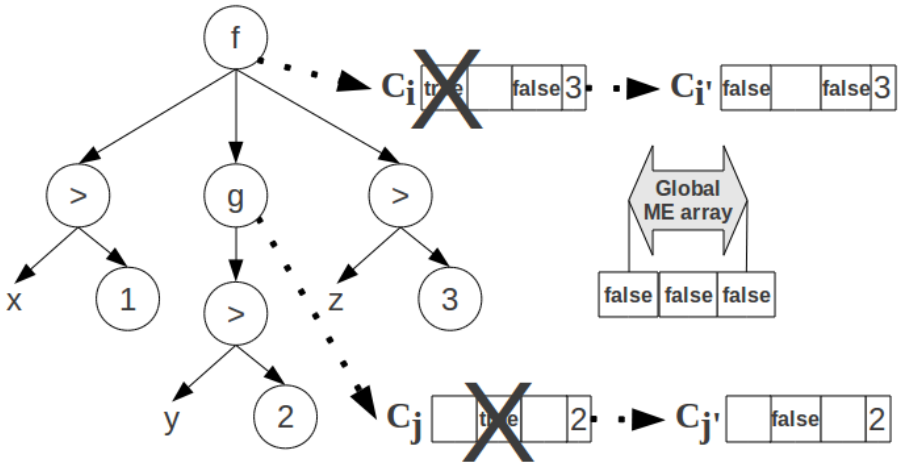

Figure 5.4: Tuples elimintated by MED

We use a global array "ALUMEVal" to implement the global ME array. In TGR the global ME array stores the value of the instance of the ME point. In Smaran the global ME array stores the value of the unreduced signature of the class of the instance of the ME point. Please see line 25, 41 in the function "ALUnormaliseG", and line 25, 43 in the function "ALUnormalise".

In the function "ALUinsertG" or "ALUinsert", MED traverses the local ME array, gets the value and compares it with the value at the corresponding location in the global array. Once MED finds two values are not equal, the tuple containing the candidate is abandoned. If MED is needed and it eliminates all candidates, the function calls the function "ALUinsert2G" to insert the instance and its candidate list to the ALU-list recycled part.

5.3 Descendants Elimination

DE cuts unnecessary matching attempts when the ascendant $s = \sigma(r|_p)$, a subterm in the instance of a RHS r, matches an LHS l. Normalization will not match the descendants of the root of s. After application of l, s rewrites to s' which probably has new descendants. Normalization tuples containing LHSs that may match the descendants of s have little chance to match the descendants of s' and thus, are removed by DE.

In some cases, normalization tuples containing descendants should *not* be deleted. When s is matching l, some variables in l are instantiated to the instances of the ALU points in s. Normalization tuples from these ALU points have to survive in DE because instances of these points remain the same in s'. For example, in Figure 5.5, a RHS r, $f(g(x,y),h(z))$, unifies with an LHS l_1, $f(x,h(z))$. $g(x,y)$ in r unifies with an LHS l_2, h(z) unifies with an LHS l_3. In the i^{th} step, the rule with r as the RHS is applied. Tuples (i,l_1,s_1), (i,l_2,s_2), (i,l_3,s_3) in which $s_1 = \sigma(f(g(x,y),h(z)))$, $s_2 = \sigma(g(x,y))$, $s_3 = \sigma(h(z))$, are built in normalization. If LRR tries the tuple

(i, l_1, s_1) before the other two tuples and s_1 matches l_1, s_1 rewrites to s'_1. In s'_1, s_2 is kept untouched because x in l_1 is instantiated to s_1. s_3 rewrites to s'_3 which is probably not equal to s_3. So the tuple (i, l_3, s_3) predicting a possible match between l_3 and s_3 should be deleted because s_3 probably does not exist in s'_1. The tuple (i, l_2, s_2) should *not* be deleted because s_2 exists in s'_1 as long as x appears in r_1, the RHS of l_1.

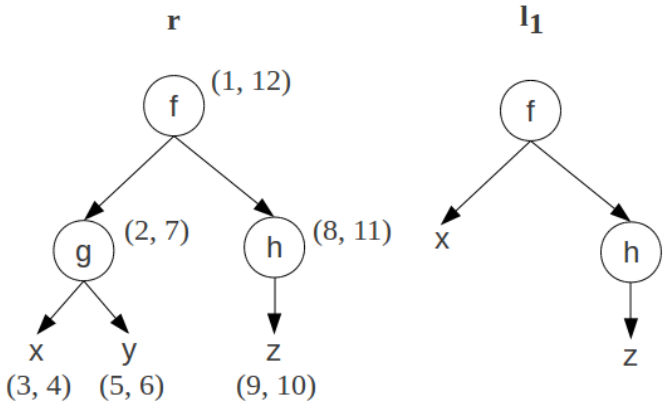

Figure 5.5: An example in DE

DE is only needed when tuples containing ascendants are popped before tuples containing descendants. This happens only when the ALU-list operates like a stack since the order to build an instance in an intermediate result is from the bottom up.

Firstly, DE needs to determine whether a subterm u is a descendant of another subterm s. We use the parenthesis theorem discussed in section 3.2.3. In the structure "RHS", we reuse field "ALU_dtime" to keep the discovery time and "ALU_ftime" to keep the finishing time. At the beginning of UP, LRR calls a function "ALUDF_RHS" to DFS every RHS to get the discovery time and the finishing time

for each subterm in the RHS. This allows us to tell whether u is a descendant of s according to the parenthesis theorem. The function "ALUDF_RHS" is very similar to the function "ALUDFS" in section 3.2.3. The former works on the expression tree implemented by the field "parameters". The latter works on the expression DAG implemented by the field "ALU_para" which keeps a dedicated copy for unification. During normalization, we extend the normalization tuple from (i, c, s) to $(i, c, s, dtime, ftime)$. In the structure "ALU_List_Node" shown in Figure 3.5, we use field "DFMin" to keep $dtime$ and "DFMax" to keep $ftime$. Take the RHS r, $f(g(x, y), h(z))$ which unifies with an LHS l_1, $f(x, h(z))$ as an example. The discovery time and finishing time are labeled in Figure 5.5.

Secondly, DE needs to determine whether u in the normalization tuple $(i, c', u,$ $dtime_1,$ $ftime_1)$ is covered by a variable instantiation. In the $j^{th}(j > i)$ step, when normalization tries to match the tuple $(i, c, s, dtime_2, ftime_2)$ where s is the ascendant of u, it checks every variable in lhs_c which instantiates to a function and keeps the $dtime$ and $ftime$ of its instantiation. u is the descendant of a variable instantiation if $dtime \leq dtime_1 \leq ftime_1 \leq ftime$. For example, in Figure 5.5, the tuple containing ascendant is $(i, l_1, f, 1, 12)$. The tuple containing descendants are $(i, l_2, g, 2, 7)$ and $(i, l_3, h, 8, 11)$. The discovery time and finishing time of the instantiation of x are (2, 7). Thus, $(i, l_2, g, 2, 7)$ is kept because it is covered by a variable instantiation and $(i, l_3, h, 8, 11)$ is deleted.

We use a global array "ALUDE" to store the $dtime$ and $ftime$ of each variable instantiation in each match and a global counter ALUDECnt for the array. Since in

TGR, variables are instantiated to a subterm implemented in the structure "signature" and the discovery time and finishing time are stored in the structure "RHS", we add two fields "ALU_dtime" and "ALU_ftime" into the structure "signature". The discovery time and finishing time are copied from the structure "RHS" to the structure "signature" in the function "ALUnormaliseG" and are then copied from the structure "signature" into the array "ALUDE" in the function "ALUnr_matchG" when a variable instantiation occurs. Similarly in Smaran, since variables are instantiated to a class, the discovery time and finishing time are copied from the unreduced signature of a class to the global array "ALUDE".

Lastly, DE eliminates normalization tuples containing descendants that are *not* covered by any variable instantiation. In the j^{th} step of normalization, upon the successful match of the normalization tuple $(i, c, s, dtime_2, ftime_2)$, DE moves along the ALU-list deleting tuples a) with same i, b) containing the descendants of s that are not covered by variable instantiations. DE stops when visiting a tuple with different i.

5.4 Same Point Elimination

SPE cuts unnecessary matching attempts after multiple normalization tuples with same s are inserted in one step of normalization. Since they are inserted in the same step, they have same i. Normalization tuples with same i and same s (defined as a group of *same point tuples*) indicate that different LHSs may match s. However, at most one LHS matches s in normalization. Once normalization finds the first tuple

that matches successfully, s rewrites to s' which is usually different from s. Thus, the remaining tuples of the group that may match s have little chance to match s' and need to be deleted.

Same points tuples result from that multiple LHSs unify with the same subterm at the same ALU point P. Thus, UP finds multiple unification pairs with same P and then normalization builds multiple tuples with same i and s. There are two reasons for multiple LHSs which unify with same subterm in a RHS. One reason is the extension of the ALU algorithm which allows multiple LHSs with different ME subterms to unify with the same subterm at same ME point in a RHS. We don't consider this situation in SPE as it can be resolved by MED. Another reason is that multiple LHS unify with each other. As long as MED is enabled, LRR needs SPE only when at least one LHS unifies with another LHS. Thus, we unify every LHS with each other at the beginning of UP. If any successful unification is found, we enable SPE. Otherwise, SPE is disabled.

To implement SPE, we add a field "Same" in the structure "ALU_List_Node" in Figure 3.5 with an initial value 0. When the function "ALUDFLM" traverses a RHS, it assigns every subterm in a RHS with a unique id which is greater than 0 and stored in the field "Same". Please see line 7, 11, 18 in the pseudocode of the function "ALUDFLM". We extend the unification pair from (C, P) to $(C, P, SAME)$ where $SAME$ keeps the value of "Same". Similarly we extend the normalization tuple from (i, c, s) to $(i, c, s, same)$. In function "ALUinsertG" or "ALUinsert", the value of $SAME$ in a unification pair is passed to $same$ in the corresponding normalization tuple. Please see line 26 in "ALUinsertG" and line 28 in function "ALUinsert".

Instead of comparing different subterms, we compare the values of "Same" to determine whether multiple unification pairs have same subterm at same point P and whether multiple normalization tuples have same s.

In either "ALUinsertG" or "ALUinsert", a group of same point tuples are inserted sequentially in the ALU-list. They are consecutive in the ALU-list. When the function "ALUpopG" or "ALUpop" pops a tuple $(i, c, s, same)$ that matches, SPE moves along the ALU-list deleting tuples with both same i and same $same$. It stops when SPE visits a tuple with either different i or different $same$. Please see line 16 in "ALUinsertG" and line 20 in "ALUinsert".

5.5 Changed Signature Detection

CSD cuts unnecessary matching attempts when Smaran and the ALU-list are working together. In Smaran, s in the normalization tuple (i, c, s) actually represents the class number of t, the instance of an ALU point. When the tuple (i, c, s) is inserted into the ALU-list in the i^{th} step, t is the unreduced signature of class s. When the tuple is popped in the j^{th} step $(j > i)$, we try to match lhs_c and the current unreduced signature of class s, t'. Since the unreduced signature of a class may change during normalization, t' is not necessarily equal to t. So lhs_c that may match t has a rare chance to match t'. If $t' \neq t$, CSD removes the tuple (i, c, s) prior to matching. Note that the direct match in section 5.1.2 does not require CSD since LRR tries to match the tuple directly in the next step. Since in TGR s is the pointer to the subterm which never changes during normalization, CSD is not needed.

To implement CSD, we extend the normalization tuple from (i, c, s) to $(i, c, s,$ $sig)$, where s store the class number and sig stores the pointer to the unreduced signature of class s when the tuple is inserted. Then we compare the latest unreduced signature with sig when the tuple is popped. If the unreduced signature changes, CSD abandons the tuple with no matching attempt. In the structure "ALU_List_Node" in Figure 3.5, we use the field "Sig" to keep sig. In the function "ALUinsert", CSD puts sig into the tuple in line 25. In the function "ALUpop", CSD compares the latest unreduced signature with sig in line 8.

5.6 The V-list

The V-list helps normalization to scan unexplored parts in intermediate terms caused by variable substitutions when the ALU-list is enabled but the DS-list option is off. Much of the overall structure of subterms in intermediate terms can be safely predicted from the RHSs. However, variable instantiations are the exceptions. Thus, we build the V-list to keep variable instantiations. The V-list, similar to the ALU list, is a singly-linked list with current pointer containing nodes pointing to variable instantiations in intermediate results. Normalization updates the list by adding new nodes when building the instances of RHSs and by popping nodes to look for matches.

When the ALU-list is empty, the V-list controls the normalization. It routes normalization to the unexplored parts so that normalization will not track all the way back to the top of current terms or intermediate results. The V-list only provides the subterms that have not been explored. Similar to the DS-list, the V-list does not

provide any candidate. Since only the subterms that have DSs at the top have chances to match, to improve efficiency, we only put pointers to variable instantiations that have DSs at the top to the V-list. Since the DS-list itself is very efficient and all nodes in the V-list points to defined subterms, the V-list is unnecessary when the DS-list is enabled. The V-list is an option with the ALU-list when the DS-list is disabled.

The V-list uses the structure "ALU_List" in Figure 3.5 for the list and the structure "ALU_List_Node" in Figure 3.4 for nodes. Figure 5.6 shows a typical V-list.

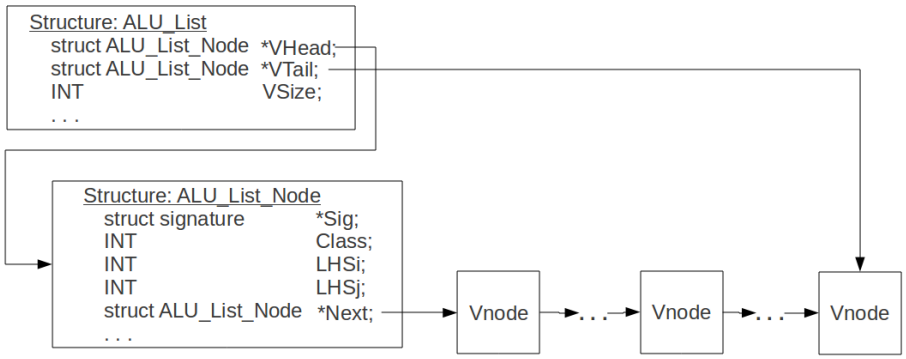

Figure 5.6: Data structure of the V-list

The V-list calls the function "ALUinsertVG" or "ALUinsertV" to insert nodes into the V-list. These two functions are simplified versions of functions "ALUinsertG" and "ALUinsert". There is no candidate list, no MED, no direct match in "ALUinsertVG" or "ALUinsertV". The pseudocodes are below.

1: **procedure** ALUINSERTVG($ALUlist, sig$)
2: initialize a node $node$
3: $node.Sig = sig$

4: $node.Next = NULL$

5: $sig.ALU\_DAV = 3$

6: $node.Next = ALUlist.VHead.Next$

7: $ALUlist.VHead.Next = node$

8: $ALUlist.VSize = ALUlist.VSize + 1$

9: **end procedure**

1: **procedure** ALUINSERTV($ALUlist, class$)

2: initialize a node $node$

3: $node.Class = class$

4: $node.Next = NULL$

5: $xClass\_List[class].ALU\_DAV = 3$

6: $node.Next = ALUlist.VHead.Next$

7: $ALUlist.VHead.Next = node$

8: $ALUlist.VSize = ALUlist.VSize + 1$

9: **end procedure**

The function "ALUpopG" or "ALUpop", when the V-list is enabled, tries to pop top node from the V-list looking for a match. It stops when a match is found or the V-list is empty. Please see line 29 in the pseudocode of "ALUpopG" or line 33 in the pseudocode of "ALUpop".

The function "ALUnormaliseG" or "ALUnormalise", when the V-list is enabled, calls the the function "ALUinsertVG" or "ALUinsertV" whenever a variable is instantiated to a defined subterm. Please see line 29, 55 in the pseudocode of "ALUnormaliseG" or line 31, 59 in the pseudocode of "ALUnormalise".

5.7 The Recyclable ALU-list

The recyclable ALU-list contains the fresh part and the recycled part. A typical recyclable ALU-list is shown in Figure 4.1. We recycle the normalization tuples that didn't match because a) tuples that didn't match before may match in the future; b) as we discussed in section 4.1 tuples are more predictive in matching than the DS-list or original methods. One reason that the subterm indicated by s in the normalization tuple (i, c, s) does not match lhs_c is that subterms of s have not been reduced. After the subterms of s are reduced, s has more chances to match lhs_c. Thus, we put tuples into the recycled part of the ALU-list instead of having the DS-list or oringinal methods to handle the subterms. There is a special case. If MED eliminates all tuples and cannot pick up any tuple, we retrieve the eliminated tuples and put them into the recycled part, which is implemented by the function "ALUinsert2G" or "ALUinsert2".

The recycled part is a singly-linked list starting after the fresh part. It simply operates as a queue. Tuples that do not match and tuples in the special case above are inserted into the recycled part. In one step of normalization, if the fresh part is empty with no successful match, LRR routes to the DS list or original methods for a match and lets the recycled part become the fresh part. Thus, in the next step, LRR gets back to the new fresh part of the ALU-list for a match.

The function "ALUinsert2G" or "ALUinsert2" copies tuples and inserts them into the recycled part of the ALU-list at the end. They are simplified version of functions "ALUinsertG" and "ALUinsert" without MED or direct match. The pseudocodes

are below.

```
1: procedure ALUINSERT2G(canlist, t, loop, min, max)
2:     candidate = canlist.Head
3:     for i = 1 → canlist.size do
4:         candidate = candidate.Next
5:         initialize a tuple tuple
6:         tuple.Sig = t
7:         tuple.LHSi = candidate.LHSi
8:         tuple.LHSj = candidate.LHSj
9:         tuple.DFMin = min
10:        tuple.DFMax = max
11:        tuple.Next = NULL
12:        add the tuple after ALUlist.Tail
13:        t.ALU_DAV = 2
14:        ALUlist.Size2 = ALUlist.Size2 + 1
15:        update ALUlist.Fresh, ALUlist.Tail if needed
16:        if t is in the DS-list then
17:            delete t from the DS-list
18:        end if
19:    end for
20: end procedure
```

```
1: procedure ALUINSERT2(canlist, t, loop, min, max)
2:     candidate = canlist.Head
```

3: $class$ = class of t ▷ no need in TGR

4: **for** $i = 1 \rightarrow canlist.size$ **do**

5: $candidate = candidate.Next$

6: initialize a tuple $tuple$

7: $tuple.Sig = t$ ▷ for CSD

8: $tuple.Class = class$ ▷ no need in TGR

9: $tuple.LHSi = candidate.LHSi$

10: $tuple.LHSj = candidate.LHSj$

11: $tuple.DFMin = min$

12: $tuple.DFMax = max$

13: $tuple.Next = NULL$

14: add the $tuple$ after $ALUlist.Tail$

15: $xClass\_List[class].ALU\_DAV = 2$ ▷ different from TGR

16: $ALUlist.Size2 = ALUlist.Size2 + 1$

17: update $ALUlist.Fresh$, $ALUlist.Tail$ if needed

18: **if** t is in the DS-list **then**

19: delete t from the DS-list

20: **end if**

21: **end for**

22: **end procedure**

5.8 Memory Management

Memory management focuses on the free lists which have been used in LRR. We introduce a free list for the candidate list, the recyclable ALU-list, and the V-list since all lists share the same data structure. We also dynamically extend the Qlist.

When LRR initializes, the function "Initialize_Structures" allocates four free lists for signatures, nodes in signature hash tree, classes (when Smaran is enabled), and nodes in Qlist. Now we add a free list of structure "ALU_List_Node". The global variable "ALUlist_free_head" is the current pointer and the global variable "ALUfreecnt" is the counter. The function "Initialize" allocates a main heap. Whenever we need a new node of structure "ALU_List_Node", we firstly check if the free list has enough free space. Secondly we check if the main heap has enough free space. If both the free list and the main heap are full, we call the system function "malloc". The pseudocode is below. We free a node by putting it immediately after the current node.

1: **if** $ALUList\_free\_head$ does not reach the tail **then**
2: $\quad ALUfreecnt = ALUfreecnt + 1$
3: $\quad newnode = ALUList\_free\_head.next$
4: $\quad ALUList\_free\_head = ALUList\_free\_head.next$
5: **else**
6: $\quad$ **if** main heap has enough free space **then**
7: $\quad\quad$ allocate *newnode* from the main heap
8: $\quad$ **else**

9: calls the system function malloc to allocate *newnode*

10: **end if**

11: **end if**

We dynamically double the length of the free list Qlist when the usage of the list reaches a threshold. The Qlist is used to store the functions with the built-in functions as the root and tempararily cannot be evaluated in normalization. When LRR builds the instance of a RHS, some built-in functions in the instance cannot be evaluated at that time. All built-in functions are inserted into the Qlist. We add two new fields into the structure "qlist". The field "len" stores the allocated length of the Qlist. The field "use" stores the used length. The function "ext_queue" doubles the length when the value "use" is greater than 85% of the value "len".

5.9 Improved DS-list

We further improve the DS-list in the following ways.

1. The DS-list is now fully working with TGR. We modify the function "DSL_reducibleG" which traverses the DS-list and calls the function "nr_reducible32G_DND" to look for matches. When the ALU-list is off, the ordinary function for normalization "normalise32G" calls "DSL_reducibleG" for a match. When the ALU-list is on, the function "ALUnormaliseG" calls "DSL_reducibleG" only if the ALU-list cannot find any match.

2. The DS-list deletes the stale subterms when LRR builds the instance of a RHS.

Previously, LRR deleted subterms that were in the DS-list but that no longer had any DS at the top when traversing the DS-list looking for matches. To have the DS-list cooperate with the ALU-list more effectively, latest LRR checks every new subterm s under the current term when building the current term and deletes it from the DS-list if it is in the DS-list and it meets any following case: a) $root(s) \notin DS$; b) when the ALU-list is on, s is the instance of an ALU point (a tuple containing s will be inserted into the ALU-list).

3. The option of approximation of innermost or outermost strategy with the DS-list is implemented. Due to sharing, strict innermost or outermost strategy cannot be implemented with either Smaran or TGR. The ALU-list with the optimizations above does not follow this innermost or outermost strategy. We can implement approximate innermost or outermost strategy with pure DS-list. The global pointer "DSL_Entry" always points to the head of the list. Since the DS-list is a circular double-linked list, the pointer prior to "DSL_Entry" is the tail of the list. We add a pointer "DSL_Active" pointing to the next inserting position. In one step of normalization, defined subterms are added behind "DSL_Active". Since the instance is built from the bottom up, the outermost defined subterms are inserted later than their descendants and located in the front of the DS-list. The DS-list in the outermost order moves current pointer to the tail to look for a match starting from the head of the DS-list while the DS-list in the innermost order moves current pointer in the opposite direction starting from the tail. "DSL_Active" moves only when the DS-list finds a match. It updates "DSL_Active" to point to the root of the active term.

5.10 Improved Statistics

We add seven fields in the structure "statistics" as counters. "ALU_unif" counts the number of unifications. "ALU_sucmatch" counts the number of successful matches predicted by the ALU-list. "ALU_ttlmatch" counts the number of total match attempts executed by the ALU-list. "DSL_sucmatch" counts the number of successful matches lead by the DS-list. "DSL_ttlmatch" counts the number of total match attempts executed by the DS-list. "ORI_sucmatch" and "ORI_ttlmatch" are the counters for the successful matches and total match attempts executed by the original reduction strategy.

Optimizations enhance the accuracy and efficiency of normalization, which will be illustrated in Chapter 6. Fast prediction, MED, MM are matured optimizations providing significant improvements. Other optimizations still have room for refinement.

Chapter 6

Results

The UP and the ALU-list in Chapter 3 and 4 and optimizations in Chapter 5 have been integrated into the LRR. The current LRR and all benchmarks discussed in this chapter can be downloaded from Dr. Verma's website `http://www.cs.uh.edu/~rmverma` and also `http:/www.cs.uh.edu/~evangui`. We also compare LRR against Maude 2.6 32-bit version without memo option which can be found at `http://maude.cs.uiuc.edu/download/` and Rascal 0.5.1 commandline version which can be found at `http://www.rascal-mpl.org/Rascal/Commandline`.

The current LRR is implemented in C and runs on Linux. Normalization time is on a 2.67GHz Intel i5 560M Ubuntu 10.10 Linux kernel 2.6.35-22 system with 8GB of memory using gcc compiler (v. 4.4.5) with optimization level 3. Normalization time depicted in the table in this chapter is the average result of 10 executions in seconds. Cases that either took more than one hour, ran out of memory, or produced an internal error show as '-'.

We use nine original benchmarks (binsort, bintree, dfa, fib, merge, qsort, rev, rfrom, sieve) to illustrate the level of efficiency. We use four new benchmarks (fact, fibb, garbage collection, lrev) in [8] for comparison. Details of all benchmarks will be discussed in Appendix. We are aware of the difficulties of comparing different software systems. Each benchmark for three systems uses exactly the same algorithm. Rules in the benchmark are semantically identical. Syntactic differences are due to differences in the rule specifications for the three systems.

Table 6.1 shows how many successful unifications are found by the UP in each original benchmarks.

Table 6.1: Unification results of all original benchmarks

Benchmarks	binsort	bintree	dfa	fib	merge	qsort	rev	rfrom	sieve
Unification No.	22	7	0	4	4	18	4	3	11

The initial terms for each original benchmark are listed in Table 6.2.

Table 6.2: Initial terms of all original benchmarks

Benchmarks	Description of Initial terms
binsort	22,000 natural numbers in random order
bintree	28,000 natural numbers in random order
dfa	100,000 functions
fib	24
merge	two lists of 200,000 consecutive natural numbers in descending order
qsort	13,000 natural numbers in random order
rev	340,000 consecutive natural numbers in reverse order
rfrom	a list of consecutive natural numbers from 340,000 to 1
sieve	a list of consecutive natural numbers from 2 to 7,700

Table 6.3 shows the normalization time in seconds for the original nine benchmarks when LRR uses TGR as the reduction method. Table 6.4 shows the normalization time for the original nine benchmarks when LRR uses Smaran as the reduction method. Bold numbers are the best results for each benchmark.

Table 6.3: Experimental results on normalization time with TGR

Benchmarks	TGR					
	Reduction No.		+ALU	+ALU +Vlist	+DSL	+DSL +ALU
binsort	1,015,360	83.0740	10.5715	10.8919	**0.3236**	0.3388
bintree	1,002,999	62.9935	19.8696	20.3201	0.3320	**0.3244**
dfa	100,000	143.4850	143.3858	143.2789	**0.0148**	0.0152
fib	300,098	23.0366	21.8914	21.9534	**21.8194**	21.9750
merge	1,000,005	0.3156	0.3084	0.3132	0.2872	**0.2676**
qsort	1,004,232	26.2384	11.8139	11.7171	0.2196	**0.1956**
rev	1,020,004	0.2896	0.2888	0.2884	0.2632	**0.2404**
rfrom	1,000,002	0.2836	0.2804	0.2800	0.2656	**0.2504**
sieve	1,011,303	0.2628	0.2772	0.2684	0.2652	**0.2552**

Table 6.4: Experimental results on normalization time with Smaran

Benchmarks	Smaran					
	Reduction No.		+ALU	+ALU +Vlist	+DSL	+DSL +ALU
binsort	995,367	147.1304	32.3380	32.2992	0.4756	**0.4508**
bintree	1,002,886	109.1112	39.1933	39.1261	0.8569	**0.8429**
dfa	100,000	193.5517	193.7437	193.9021	**6.5424**	6.5480
fib	50	0.0000	0.0000	0.0000	0.0000	0.0000
merge	1,000,005	0.9985	0.9761	0.9757	**0.8413**	0.8805
qsort	1,000,603	142.6425	73.9294	74.1002	14.2737	**14.2485**
rev	1,020,004	0.8481	0.8273	0.8297	0.8245	**0.8221**
rfrom	1,000,002	1.0301	1.0325	1.0365	**0.9457**	0.9597
sieve	1,004,582	0.7416	0.7248	0.7300	0.7096	**0.6812**

Table 6.3 and 6.4 show following. a) The combination of the ALU-list and the DS-list performs best in most benchmarks. For the remaining benchmarks, the DS-list alone is the best. However, the combination of the ALU-list and the DS-list is close enough. b) dfa has zero unification, thus, the ALU-list does not help at all. The best way to normalize the initial term is to start from the innermost function. The default order of the DS-list is innermost, so the DS-list is the fastest strategy for dfa. c) Smaran cuts the reduction number of fib from 300,098 to 50, which improves the normalization significantly from above 20 seconds to almost 0. Smaran can calculate $fib(100,000)$ in less than 600 seconds with 200,002 reductions. d) In most cases, the DS-list alone outperforms the original strategy. The ALU-list alone also performs better than the original strategy.

Table 6.5 and Table 6.6 show the percentages of successful matches that are found by the ALU-list, the DS-list and the original strategies in original benchmarks and accuracy. The percentage is calculated as $\frac{the\ number\ of\ the\ successful\ matches}{the\ number\ of\ the\ total\ reductions}$. The accuracy is calculated as $\frac{the\ number\ of\ the\ successful\ matches}{the\ number\ of\ the\ total\ match\ attempts}$. "-" means $the\ number\ of\ the\ \ successful\ matches = 0$ and thus, accuracy does not need to be calculated. In Table 6.5 and Table 6.6, B stands for benchmarks, RS stands for reduction strategy A stands for accuracy and P stands for percentage.

B	RS	TGR		TGR+ALU		TGR+ALU +Vlist		TGR+DSL		TGR+ALU +DSL	
		A(%)	P(%)	A(%)	P(%)	A(%)	P(%)	A(%)	P(%)	A(%)	P(%)
binsort	by ALU	-	-	100.00	94.41	98.54	96.11	-	-	100.00	97.24
	by DSL	-	-	-	-	-	-	100.00	100.00	3.80	2.76
	by ORI	0.42	100.00	38.96	5.59	35.53	3.89	-	-	-	-
bintree	by ALU	-	-	100.00	97.21	100.00	97.21	-	-	100.00	97.21
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	2.79
	by ORI	0.26	100.00	0.01	2.79	0.01	2.79	-	-	-	-
dfa	by ALU	-	-	-	-	-	-	-	-	-	-
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	100.00
	by ORI	0.01	100.00	0.01	100.00	0.01	100.00	-	-	-	-
fib	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	0.00
	by ORI	100.00	100.00	100.00	0.00	100.00	0.00	-	-	-	-
merge	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	0.00
	by ORI	100.00	100.00	60.00	0.00	60.00	0.00	-	-	-	-
qsort	by ALU	-	-	100.00	91.34	95.66	91.34	-	-	100.00	96.12
	by DSL	-	-	-	-	-	-	100.00	100.00	82.01	3.88
	by ORI	73.12	100.00	27.92	8.66	27.92	8.66	-	-	-	-
rev	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	0.00
	by ORI	100.00	100.00	75.00	0.00	100.00	0.00	-	-	-	-
rfrom	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	0.00
	by ORI	100.00	100.00	100.00	0.00	100.00	0.00	-	-	-	-
sieve	by ALU	-	-	99.90	99.90	99.90	99.90	-	-	99.90	99.90
	by DSL	-	-	-	-	-	-	100.00	100.00	100.00	0.10
	by ORI	99.90	100.00	99.90	0.10	99.90	0.10	-	-	-	-

Table 6.5: Results on accuracy and percentage of successful matches with TGR

B	RS	Smaran		Smaran+ALU		Smaran+ALU +Vlist		Smaran+DSL		Smaran+ALU +DSL	
		A(%)	P(%)	A(%)	P(%)	A(%)	P(%)	A(%)	P(%)	A(%)	P(%)
binsort	by ALU	-	-	100.00	94.30	98.26	96.29	-	-	100.00	97.99
	by DSL	-	-	-	-	-	-	94.19	100.00	99.98	2.01
	by ORI	0.41	100.00	38.96	5.70	34.84	3.71	100.00	0.00	-	-
bintree	by ALU	-	-	100.00	97.21	100.00	97.21	-	-	100.00	97.21
	by DSL	-	-	-	-	-	-	97.28	100.00	100.00	2.79
	by ORI	0.26	100.00	0.01	2.79	0.01	2.79	-	-	-	-
dfa	by ALU	-	-	-	-	-	-	-	-	-	-
	by DSL	-	-	-	-	-	-	49.99	100.00	49.99	100.00
	by ORI	0.01	100.00	0.01	100.00	0.01	100.00	0.00	0.00	0.00	0.00
fib	by ALU	-	-	100.00	98.00	100.00	98.00	-	-	100.00	98.00
	by DSL	-	-	-	-	-	-	100.00	98.00	-	-
	by ORI	100.00	100.00	100.00	2.00	100.00	2.00	100.00	2.00	100.00	2.00
merge	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	66.67	0.00
	by ORI	100.00	100.00	60.00	0.00	60.00	0.00	50.00	0.00	50.00	0.00
qsort	by ALU	-	-	100.00	91.75	100.00	91.75	-	-	100.00	95.51
	by DSL	-	-	-	-	-	-	91.22	100.00	52.85	4.49
	by ORI	68.54	100.00	28.17	8.25	28.17	8.25	100.00	0.00	-	-
rev	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	66.67	0.00
	by ORI	100.00	100.00	75.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
rfrom	by ALU	-	-	100.00	100.00	100.00	100.00	-	-	100.00	100.00
	by DSL	-	-	-	-	-	-	100.00	100.00	-	-
	by ORI	100.00	100.00	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
sieve	by ALU	-	-	99.90	99.90	99.90	99.90	-	-	99.90	99.90
	by DSL	-	-	-	-	-	-	99.90	100.00	100.00	0.10
	by ORI	99.90	100.00	99.90	0.10	99.90	0.10	50.00	0.00	50.00	0.00

Table 6.6: Results on accuracy and percentage of successful matches with Smaran

Table 6.5 and Table 6.6 show that a) When the ALU-list is enabled, it leads to more than 90% of the successful matches. b) The accuracy of the ALU-list is 100% in most cases and above 95% in the remaining cases. c) All nine original benchmarks can be divided into three groups as following.

1. This group contains one benchmark dfa. UP returns no unification. The ALU-list reduction strategy alone is not helpful. In Table 6.5 and 6.6, the ALU-list contributes no successful match and in Table 6.3 and 6.4, normalization time of the ALU-list only and the V-list is comparative to the time of the original strategy. When the DS-list is enabled, the default innermost strategy enhances the accuracy of finding the next match from 0.01% to 49.99% with Smaran and 100% with TGR. Thus, the DS-list cuts the normalization time sharply. Also, with the DS-list, TGR finds the next match more accurately than Smaran, so TGR spends less time in normalization than Smaran does.

2. This group contains benchmarks binsort, bintree, and qsort. The DS-list or the ALU-list effectively increases the accuracy in finding the next match. In binsort and bintree, the accuracy boosts from less than 1% to nearly 100%. Thus, the normalization time of the DS-list or the ALU-list deceases significantly. In binsort, the V-list finds more successful matches than the ALU-list alone though with lower accuracy. Thus, the normalization time of the ALU-list and the V-list are close. Also, in binsort when TGR, the ALU-list and the DS-list are enabled, the accuracy of the DS-list drops sharply to 3.80%. Thus, the best result comes from the TGR and the DS-list only.

3. This group contains the remaining benchmarks fib, rev, rfrom, sieve. The DS-list or the ALU-list does not or minimally increases the accuracy due to the fact that the accuracy of original strategies is close to 100%. The improvement of the DS-list or the ALU-list is subtle.

Table 6.7 shows the normalization time for the four new benchmarks.

System	factorial		fibonacci			garbage collection		list reversal	
	(8)	(10)	(20)	(25)	(30)	(4,4,2,0,1)	(4,4,4,0,1)	(1000)	(10000)
Maude32	0.004	0.408	0.009	0.129	1.750	0.000	0.363	0.039	3.652
Rascal	0.122	-	0.299	-	-	0.012	-	1.351	153.010
Smaran	0.037	6.524	0.010	0.118	2.215	0.000	0.907	3.729	-
Smaran+DSL	0.028	5.252	0.010	0.111	2.140	0.001	0.824	0.200	-
Smaran+ALU	0.030	5.516	0.010	0.116	2.289	0.001	0.969	0.252	-
Smaran+DSL+ALU	0.026	4.958	0.009	0.108	2.078	0.000	0.802	0.191	-
TGR	0.011	0.851	2.066	1029.072	-	0.000	0.280	2.777	-
TGR+DSL	0.007	0.844	0.025	0.354	5.318	0.000	0.983	0.120	-
TGR+ALU	0.006	0.821	0.727	357.138	-	0.000	0.843	0.124	-
TGR+DSL+ALU	0.006	0.714	0.039	0.614	9.681	0.000	0.779	0.108	-

Table 6.7: Experimental results on normalization time

From Table 6.7, though we find that Maude without memo is the fastest option in most benchmarks, Smaran and/or TGR are close. It is interesting to see that Smaran is not far behind even in examples that do not use history, despite saving the entire history of rule applications. Rascal runs slowly in most cases. The latest ALU-list and the latest DS-list alone beats TGR or Smaran in most cases. We noticed that in list reversal, either the ALU-list or the DS-list improves the efficiency sharply. The combination of the ALU-list and the DS-list always performs better than Smaran and performs better than TGR in half of the cases.

Much of the time in normalization is spent in looking for the next match. The ALU-list and the DS-list can reduce this time by predicting the next match more precisely. The ALU-list cuts the unnecessary matching attempts significantly. Although it does not yet control normalization independently, the percentage of successful matches is relatively high.

Chapter 7

Conclusion

We have presented UP, a preprocessor for rules based on unification algorithm in Chapter 3, and the ALU-list, a new reduction strategy in Chapter 4. We have also discussed a number of optimizations related to UP, the ALU-list and the earlier LRR in Chapter 5. The results in Chapter 6 prove that these work together transcends the previous version of LRR in a) cutting the time spent in traversing both the term and the rules in order to find a match, b) dominating ($> 90\%$) the process that locates the next match and predicting the next match in great accurarcy($> 95\%$).

7.1 Future Work

LRR still has room for improvement. We are working on following ideas.

- UP can be extended to unify RHSs with subterms of LHSs. This would allow to

find matches at points that failed in previous attempts, if there was a successful reduction below that point.

- Deletion of the DS-list needs to be refined. Similar to DE in section 5.3, nodes in the DS-list that are covered by variable instantiations should not be deleted.

- A more powerful analyzing tool needs to be implemented. Current statistics collection can be extended to include reasons for matching failure, counters of all optimizations, and so on.

Bibliography

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[2] Leo Bachmair, C. R. Ramakrishnan, I. V. Ramakrishnan, and Ashish Tiwari. Normalization via rewrite closures. In *RTA*, pages 190–204, 1999.

[3] Peter Borovanskỳ, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélene Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. Elan v3.4 user manual. *LORIA, Nancy (France)*, 567, 2000.

[4] C. Choppy. *ASSPEGIQUE user's manual*. Université de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique, 1988.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. The Maude 2.0 system. In *RTA*, pages 76–87, 2003.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 0-262-03384-4 edition, 2009.

[8] Francisco Durán, Manuel Roldán, Jean-Christophe Bach, Emilie Balland, Mark van den Brand, James R. Cordy, Steven Eker, Luc Engelen, Maartje de Jonge, and Karl Trygve Kalleberg. The third rewrite engines competition. In *WRLA*, pages 243–261, 2010.

[9] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of obj2. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[10] Alfons Geser and Heinrich Hußmann. Experiences with the RAP system - a specification interpreter combining term rewriting and resolution. In *ESOP*, pages 339–350, 1986.

[11] Alfons Geser, Heinrich Hußmann, and Andreas Mück. A compiler for a class of conditional term rewriting systems. In *CTRS*, pages 84–90, 1987.

[12] J. Goguen, C. Kirchner, and J. Meseguer. Concurrent term rewriting as a model of computation. In *Graph Reduction*, pages 53–93. Springer, 1987.

[13] Joseph A Goguen and Grant Malcolm. *Software Engineering with OBJ: Algebraic Specification in Action*, volume 2. Springer, 2000.

[14] Joseph A Goguen and José Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[15] M. Hermann, C. Kirchner, and H. Kirchner. Implementations of term rewriting systems. *The Computer Journal*, 34(1):20–33, 1991.

[16] Gerard Huet and Derek C Oppen. Equations and rewrite rules. *Formal Language Theory: Perspectives and Open Problems*, pages 349–405, 1980.

[17] José Meseguer. Membership algebra as a logical framework for equational specification. In *WADT*, pages 18–61, 1997.

[18] S. Narain. Log (f): An optimal combination of logic programming, rewriting, and lazy evaluation. Technical report, DTIC Document, 1988.

[19] Nicholas Radcliffe and Rakesh M. Verma. Uniqueness of normal forms is decidable for shallow term rewrite systems. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 284–295, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[20] Tony Rush and Derek Coleman. Architecture for conditional term rewriting. In *CTRS*, pages 266–278, 1987.

[21] Mark van den Brand. Applications of the Asf+Sdf meta-environment. In *GTTSE*, pages 278–296, 2005.

[22] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[23] Rakesh Verma and Shalitha Senanayake. LR^2 : A laboratory for rapid term graph rewriting. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 252–255, 1999.

[24] Rakesh Verma and James Thigpen. Data structures for fast normalization and strategies. In *Proceedings of 4th International Workshop on Reduction Strategies in Rewriting and Programming*, pages 45–49, 2004.

[25] Rakesh M. Verma. Smaran: A congruence-closure based system for equational computations. In *Proceedings of the International Conference on Automated Deduction*, pages 457–461, 1993.

[26] Rakesh M. Verma. Static analysis techniques for equational logic programming. *CoRR*, cs.LO/0010034, 2000.

[27] Rakesh M Verma and Wei Guo. Does unification help in normalization? *UNIF 2011*, page 52, 2011.

[28] R.M. Verma. A theory of using history for equational systems with applications. *Journal of the ACM (JACM)*, 42(5):984–1020, 1995.

[29] W. Yu and R.M. Verma. Visualization of rule-based programming. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 1258–1259. ACM, 2008.

Appendix

Benchmarks

The rules of the nine original benchmarks and four new benchmarks for LRR are listed below.

1. binsort. This program sorts a list by inserting values into a binary search tree.

$$ins(x, nil) \Rightarrow node(nil, x, nil) \quad (7.1)$$

$$ins(x, node(l, v, r)) \Rightarrow instest(x, >(x, v), <(x, v), l, v, r)) \quad (7.2)$$

$$instest(x, false, true, l, v, r) \Rightarrow node(ins(x, l), v, r) \quad (7.3)$$

$$instest(x, true, false, l, v, r) \Rightarrow node(l, v, ins(x, r)) \quad (7.4)$$

$$instest(x, false, false, l, v, r) \Rightarrow node(l, v, r) \quad (7.5)$$

$$cat(:(x, y), z) \Rightarrow :(x, cat(y, z)) \quad (7.6)$$

$$cat(nil, z) \Rightarrow z \quad (7.7)$$

$$binsort(:(x, y)) \Rightarrow bs(ins(x, nil), y) \quad (7.8)$$

$$bs(n, :(x, y)) \Rightarrow bs(ins(x, n), y) \quad (7.9)$$

$$bs(n, nil) \Rightarrow makelist(n) \quad (7.10)$$

$$makelist(node(l, v, r)) \Rightarrow cat(makelist(l), :(v, makelist(r))) \quad (7.11)$$

$$makelist(nil) \Rightarrow nil \quad (7.12)$$

2. bintree. This program inserts a value into a binary search tree.

$$ins(x, nil) \Rightarrow node(nil, x, nil) \quad (7.13)$$

$$ins(x, node(l, v, r)) \Rightarrow instest(x, >(x, v), <(x, v), l, v, r)) \quad (7.14)$$

$$instest(x, false, true, l, v, r) \Rightarrow node(ins(x, l), v, r) \quad (7.15)$$

$$instest(x, true, false, l, v, r) \Rightarrow node(l, v, ins(x, r)) \quad (7.16)$$

$$instest(x, false, false, l, v, r) \Rightarrow node(l, v, r) \quad (7.17)$$

3. dfa. This program simulates a deterministic finite automaton.

$$a(q0) \Rightarrow q1 \tag{7.18}$$

$$b(q0) \Rightarrow q0 \tag{7.19}$$

$$a(q1) \Rightarrow q0 \tag{7.20}$$

$$b(q1) \Rightarrow q1 \tag{7.21}$$

4. fib. This program calculates the n^{th} Fibonacci numbers.

$$fib(x) \Rightarrow f(>(x,1),x) \tag{7.22}$$

$$f(true,x) \Rightarrow +(fib(-(x,1)),fib(-(x,2))) \tag{7.23}$$

$$f(false,x) \Rightarrow 1; \tag{7.24}$$

5. merge. This program merges two lists into one.

$$merge(nil,nil) \Rightarrow nil \tag{7.25}$$

$$merge(:(x,y),nil) \Rightarrow :(x,y) \tag{7.26}$$

$$merge(nil,:(x,y)) \Rightarrow :(x,y) \tag{7.27}$$

$$merge(:(x,y),:(u,v))) \Rightarrow :(x,:(u,merge(y,v))) \tag{7.28}$$

6. qsort. This program implements quicksort on a list of natural numbers.

$$cat(:(x,y),z) \Rightarrow :(x,cat(y,z)) \tag{7.29}$$

$$cat(nil,z) \Rightarrow z \tag{7.30}$$

$$sort(nil) \Rightarrow nil \tag{7.31}$$

$$sort(:(x,y)) \Rightarrow cat(sort(smaller(x,y)),:(x,sort(larger(x,y)))) \tag{7.32}$$

$$smaller(x,nil) \Rightarrow nil \tag{7.33}$$

$$smaller(x,:(y,z)) \Rightarrow f(<(x,y),x,y,z) \tag{7.34}$$

$$f(true,x,y,z) \Rightarrow smaller(x,z) \tag{7.35}$$

$$f(false,x,y,z) \Rightarrow :(y,smaller(x,z)) \tag{7.36}$$

$$larger(x,nil) \Rightarrow nil \tag{7.37}$$

$$larger(x,:(y,z)) \Rightarrow g(<(x,y),x,y,z) \tag{7.38}$$

$$g(true,x,y,z) \Rightarrow :(y,larger(x,z)) \tag{7.39}$$

$$g(false,x,y,z) \Rightarrow larger(x,z) \tag{7.40}$$

7. rev. This program reverses a list.

$$rev(x) \Rightarrow apprev(x,nil) \tag{7.41}$$

$$apprev(:(x,y),z) \Rightarrow apprev(y,:(x,z)) \tag{7.42}$$

$$apprev(nil,w) \Rightarrow w \tag{7.43}$$

8. rfrom. This program outputs a list of natural numbers in a reverse order.

$$rfrom(x, y) \Rightarrow rffrom(>(y, 0), x, y) \tag{7.44}$$

$$rffrom(true, x, y) \Rightarrow :(x, rfrom(-(x, 1), -(y, 1))) \tag{7.45}$$

$$rffrom(false, x, y) \Rightarrow nil \tag{7.46}$$

9. sieve. This program outputs a list of prime numbers from a list of natural numbers greater than 1.

$$fsieve(true, x, l, y) \Rightarrow :(x, sieve(filter(x, l), -(y, 1))) \tag{7.47}$$

$$fsieve(false, x, l, y) \Rightarrow nil \tag{7.48}$$

$$filter(n, :(x, l)) \Rightarrow ffilt(=(\%(x, n), 0), n, x, l) \tag{7.49}$$

$$filter(n, nil) \Rightarrow nil \tag{7.50}$$

$$ffilt(true, n, x, l) \Rightarrow filter(n, l) \tag{7.51}$$

$$ffilt(false, n, x, l) \Rightarrow :(x, filter(n, l)) \tag{7.52}$$

$$sieve(:(x, l), y) \Rightarrow fsieve(>(y, 0), x, l, y) \tag{7.53}$$

$$sieve(nil, y) \Rightarrow nil \tag{7.54}$$

$$sieve(x, 0) \Rightarrow nil \tag{7.55}$$

10. fact. This program calculates the factorial of a natural number.

$$plus(0, n) \Rightarrow n \tag{7.56}$$

$$plus(s(n), m) \Rightarrow s(plus(n, m)) \tag{7.57}$$

$$times(0, n) \Rightarrow 0 \tag{7.58}$$

$$times(s(n), m) \Rightarrow plus(m, times(n, m)) \tag{7.59}$$

$$fact(0) \Rightarrow s(0) \tag{7.60}$$

$$fact(s(n)) \Rightarrow times(s(n), fact(n)) \tag{7.61}$$

11. fibb. This program calculates the n^{th} Fibonacci numbers.

$$plus(0, n) \Rightarrow n \tag{7.62}$$

$$plus(s(n), m) \Rightarrow s(plus(n, m)) \tag{7.63}$$

$$fibb(0) \Rightarrow s(0) \tag{7.64}$$

$$fibb(s(0)) \Rightarrow s(0) \tag{7.65}$$

$$fibb(s(s(n))) \Rightarrow plus(fibb(s(n)), fib(n)) \tag{7.66}$$

12. garbage collection.

$$c(zero, y) \Rightarrow y \tag{7.67}$$

$$c(s(x), y) \Rightarrow s(c(x, y)) \tag{7.68}$$

$$f1(x, y, z, t, u) \Rightarrow f2(x, y, z, y, z, t, u) \tag{7.69}$$

$$f2(x, y, s(z), n, p, t, u) \Rightarrow f2(x, y, z, n, p, c(t, t), u) \tag{7.70}$$

$$f2(x, s(y), zero, n, p, t, u) \Rightarrow f2(x, y, p, n, p, t, t) \tag{7.71}$$

$$f2(s(x), zero, zero, n, p, t, u) \Rightarrow f2(x, n, p, n, p, s(zero), zero) \tag{7.72}$$

$$f2(zero, zero, zero, n, p, t, u) \Rightarrow t \tag{7.73}$$

13. lrev. This program reverses a list.

$$plus(0, n) \Rightarrow n \tag{7.74}$$

$$plus(s(n), m) \Rightarrow s(plus(n, m)) \tag{7.75}$$

$$times(0, n) \Rightarrow 0 \tag{7.76}$$

$$times(s(n), m) \Rightarrow plus(m, times(n, m)) \tag{7.77}$$

$$gen(s(n)) \Rightarrow :(s(n), gen(n)) \tag{7.78}$$

$$gen(0) \Rightarrow :(0, nil) \tag{7.79}$$

$$conc(:(e, t), tt) \Rightarrow :(e, conc(t, tt)) \tag{7.80}$$

$$conc(nil, tt) \Rightarrow tt \tag{7.81}$$

$$rev(:(e, t)) \Rightarrow conc(rev(t), :(e, nil)) \tag{7.82}$$

$$rev(nil) \Rightarrow nil \tag{7.83}$$